Institut für Informatik
der Technischen Universität München

Dynamic Migration of Object Semantics among Heterogeneous Environments

Michael P. Wagner

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitzender:

Univ.-Prof. Dr. Christoph Zenger

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Johann Schlichter

2. Univ.-Prof. Bernd Brügge, Ph.D.

Die Dissertation wurde am 7.11.2001 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 8.5.2001 angenommen.

Abstract

The migration of software entities between information systems has been studied mainly in the context of process migration between nodes of distributed operating systems and to lesser extent as object migration between programming environments. Most existing approaches assume homogeneous systems as a prerequisite to migration, despite the fact that heterogeneous hardware, operating systems and programming languages are the norm in most practical environments today.

This work is concerned with migration in the context of heterogeneity and focuses on the migration of atomic objects between heterogeneous language environments. A characterization of object migration provides for the first time a coherent context for the comparison of different existing migration systems and discusses the possible levels of and alternative approaches to heterogeneous migration.

This analysis leads to the development of a novel migration mechanism capable of transferring the state and behavior of atomic language objects between heterogeneous environments using only limited knowledge about the destination environments. The mechanism migrates a set of related objects as well as their semantics.

The design of this mechanism called Heterogeneous Language Migration (HLM) takes a pragmatic approach guided by a set of objectives and is restricted to a defined set of language concepts. Details of its architecture, abstractions, algorithms, representation formats and programming interfaces are described and discussed. The feasibility of this approach is shown in a prototypical implementation and demonstrated in a working example.

In contrast to other related approaches, as for example the Java application environment, the HLM migration mechanism is able to handle not only different hardware and operating system platforms but also heterogeneous language environments as well as differences of libraries and applications.

The HLM migration mechanism is intended to work without changes to the definitions of objects during migration and without changes to the programming environments involved prior to migration. As a consequence applications need to be designed for migration. Tools that aid in the development of applications for this migration mechanism are proposed.

The range of enhancements that can be used to augment the HLM migration mechanism is discussed as well as possible extensions that diverge from the objectives of the pragmatic approach. The necessary support for additional language concepts and the applicability of the HLM migration mechanism to existing language environments are discussed as well.

Some examples of possible applications of heterogeneous object migration illustrate the potential advantages of the HLM migration mechanism in practical situations and innovative usage areas. A summary of the achieved results and a perspective of promising future research directions conclude this work.

Kurzfassung

Die Migration von Softwareeinheiten zwischen Informationssystemen wurde bislang hauptsächlich im Kontext von Prozessmigration zwischen Knotenrechnern verteilter Betriebssysteme und in geringerem Umfang als Objektmigration zwischen Anwendungsumgebungen untersucht. Die meisten bestehenden Ansätze bedingen homogene Systeme als eine Voraussetzung für die Migration, ungeachtet der Tatsache, dass heterogene Hardware, Betriebssysteme und Programmiersprachen in den meisten praktischen Umgebungen heute vorherrschen.

Die vorliegende Arbeit beschäftigt sich mit Migration im Kontext von Heterogenität und konzentriert sich auf die Migration von atomaren Objekten zwischen heterogenen Sprachumgebungen. Eine Charakterisierung der Objektmigration bereitet erstmalig einen kohärenten Kontext für den Vergleich der verschiedenen existierenden Migrationssysteme und diskutiert die möglichen Ebenen sowie die alternativen Ansätze für eine heterogene Migration.

Diese Analyse führt zur Entwicklung eines neuen Migrationsmechanismus der in der Lage ist, den Zustand und das Verhalten von atomaren Sprachobjekten zwischen heterogenen Umgebungen zu übertragen mit nur beschränktem Wissen über die Zielumgebungen. Der Mechanismus migriert eine Menge zusammengehöriger Objekte sowie deren Semantik.

Das Design dieses Mechanismus der Heterogeneous Language Migration (HLM) genannt wird folgt einem pragmatischen Ansatz der von einigen Vorgaben geleitet und auf eine definierte Menge von Sprachkonzepten beschränkt ist. Details der Architektur, der Abstraktionen, der Algorithmen, der Repräsentationsformate und der Programmierschnittstellen werden beschrieben und diskutiert. Die Funktionsfähigkeit dieses Ansatzes wird anhand einer prototypischen Implementierung gezeigt und an einem praktischen Beispiel demonstriert.

Im Gegensatz zu verwandten Ansätzen wie zum Beispiel der Java Anwendungsumgebung ist der HLM Migrationsmechanismus nicht nur in der Lage mit unterschiedlicher Hardware und Betriebssystemen umzugehen, sondern auch mit heterogenen Sprach- und Ausführungsumgebungen sowie mit Unterschieden Bibliotheken und Applikationen.

Der HLM Migrationsmechanismus wurde konzipiert, um ohne Änderungen der Definition der Objekte während der Migration beziehungsweise der involvierten Programmierumgebungen vor der Migration zu arbeiten. Als Konsequenz müssen Anwendungen auf die Migration zugeschnitten werden. Werkzeuge, die eine Entwicklung von Anwendungen des Migrationsmechanismus unterstützen, werden vorgeschlagen.

Das Spektrum von Ansätzen den Migrationsmechanismus zu verbessern wird diskutiert ebenso wie mögliche Erweiterungen die von den Vorgaben des pragmatischen Ansatzes abweichen. Die notwendige Unterstützung für zusätzliche Sprachkonzepte und die Anwendbarkeit des HLM Migrationsmechanismus in bestehenden Sprachumgebungen werden ebenfalls diskutiert.

Einige Beispiele möglicher Anwendungen der heterogenen Objektmigration illustrieren die potentiellen Vorteile des HLM Migrationsmechanismus in praktischen Situationen und innovativen Einsatzgebieten. Eine Zusammenfassung der Ergebnisse und ein Ausblick auf vielversprechende zukünftige Forschungsansätze runden das Werk ab.

Preface

I would like to thank Professor Dr. Johann Schlichter for the opportunity to conduct this research as an external doctorate, for his constant support, encouragement, patience and trust. I am also greatly indebted to Professor Bernd Brügge, Ph.D. for his constructive criticism and comments during the preparation of the manuscript.

Furthermore, I am grateful to all colleagues of the Chair for Applied Informatics - Cooperative Systems, especially Uwe Borghoff, Martin Bürger, Tristan Daude, Stefan Herman, Michael Koch, Daniel Koehler and Gunnar Teege for fruitful discussions as well as Elfriede Bunke for finding a solution to every technical problem and Evelyn Gemkow for the administrative support.

I owe Alessandro Tonchia for unknowingly getting me out of my devastating thesis-writing-deadlock, Michael Karger and Peter Maegdefrau for helping me set the preconditions for finishing this work and Stefan Herman for some essential organizational and technical tips during the final preparation of the manuscript.

Finally, I have to thank my family for their enduring support and patience, my parents without which I would not have been able to attempt this work, my sister Susanne for identifying all the germanisms in an earlier manuscript (so all remaining typos are mine ;-), my two sons Marcus and Niclas to which I have to apologize for being too busy for far too long and most importantly my wife Rita - without her love and belief in me I would never have conquered this task.

Michael P. Wagner

Munich, 4th of July 2001

Table of Contents

1 Introduction

Among the major developments that characterize the current state of the information technology two trends stand out as long lasting and wide spread: the growing distribution of hard- and software systems and the increasing usage of object technologies. Distributed systems range from large commercial systems to small portable and personal devices that communicate through a multitude of interconnections from local networks to the global Internet based on telecommunication backbones as well as to pervasive wireless services.

Object technologies provide abstractions that ease the design, implementation and maintenance of software systems in the small and in the large. Despite the development of several quite different “schools” of object technology, some common concepts can be identified. Concepts like encapsulation and message passing do not only simplify the construction of software systems but also match the nature of distributed systems and are increasingly used to implement distributed systems.

Distribution

The advances of computer networks have led to the development of sophisticated mechanisms of sharing across distributed systems. From file and database access techniques to generalized message passing mechanisms a number of alternatives are available to the designer of distributed systems. The implementation technologies used, reach from simple network protocols to advanced remote method invocation mechanisms.

In recent years, the near exponential growth of the Internet has widened the use of distributed systems. The commercial interest in a global information infrastructure as well as in wireless communication networks has opened up new ways of thinking about the construction of distributed systems.

Distribution is no longer viewed as an exception of otherwise local computations but rather perceived as a normal mode of operation for an application if not as an opportunity for innovation. Increasingly systems are designed for distributed use and new technologies are developed for their support as for example autonomous software agents.

Objects

The object technologies on the other hand have advanced considerably since their inception in the late sixties and seventies. Today the principles of object technology are applied not only as part of various programming languages but also as the foundation of design methodologies for the construction of large software systems.

As a consequence the reach of object technology has expanded to the whole range of information technology - from operating systems through database management systems to specialized application software. Some principles of object technology as for example encapsulation or message passing represent the state of the art of today’s software systems.

Object technology itself has developed into several fractions from orthodox object-oriented systems through hybrid extensions of existing programming environments to innovative forms like environments with prototypes or active objects. Component based software environments extend object technology through the grouping of language objects to components that can be easily combined to implement higher-level services.

Objects and Distribution

Designers of distributed software systems increasingly take advantage of the principles of object technologies. The encapsulation of the state and behavior of objects as well as the passing of messages between objects are very similar to the computer node and network communication “nature” of distributed systems. Object technologies are therefore increasingly used to master the growing complexity of distributed systems.

Problems of distribution are generally addressed through the implementation of some form of transparency, effectively rendering the management of distribution invisible for the application developer and user. While application programs that are independent of distribution can be designed more easily, certain aspects of distribution cannot be overcome through transparencies. The various software components of distributed systems stay where they are and especially low bandwidth connections remain perceivable.

Despite the time lags that set remote operations apart from local ones, full transparency of distribution may not be desirable in all cases. Some applications will need to know where objects reside. E.g., awareness of distribution is essential for mobile systems that can be subject to intentional or accidental disconnections from stationary networks.

The use of object technologies for the construction of distributed systems is in most cases confined to the use of remote message passing mechanisms, usually implemented through the technique of proxy objects. Although used in an object based context remote method invocation (RMI) resembles mainly the concept of remote procedure calls (RPC) borrowed from distributed programming in procedural systems.

While RMI extends the local message passing mechanism through the transfer of messages between distributed systems, its implementation through proxy objects hides differences between local and remote message passing since a local proxy acts on behalf of the remote object it represents. With RMI the distribution of objects is just hidden and can only be influenced through configuration prior to the execution of the various components of the distributed systems.

Remote method invocation establishes communication between distributed software-systems and proxy objects aid to make distribution transparent to the programmer and user of these systems. However, both techniques fail to join objects and distribution into one seamless notion. The separation of systems imposed by distribution is not overcome as remote objects can be referenced and their methods can be invoked but all objects remain at their locations.

Migration

A different approach to design distributed systems is taken by a technology called migration. This technology addresses the most fundamental problem of distribution, the locality of software entities by transferring data, of functionality and of computations between distributed systems. Transparency techniques blur the distinction between local and remote objects but only move the flow of control to the new point of execution and return results locally.

The migration technology separates data, functionality and computations from specific locations and moves it to distributed resources as necessary. The best-known migration technology - process migration - for example is often used to balance the load between systems through the transfer of whole processes from systems with high load to those with less load [Nut1994a].

While process migration mechanisms achieve their respective objectives, the applicability of this technique as a general software design principle is limited as it works only for comparatively coarse-grained processes. It is difficult to implement as existing operating system software has to be modified extensively and processes can only be effectively transferred between nodes of the same operating system and hardware.

Object Migration

Given the fundamental concept of encapsulation, inherent to object technology, the transfer of independent objects between distributed systems appears to be a more natural way to combine distribution and objects. This alternative technology called object mobility or object migration[1] implements migration at a finer grained levels of abstraction. Remote resources are not made available to local objects; instead, objects are moved to the location of resources.

Object migration does not substitute remote method invocation as a communication technique. It rather adds a complete new way to construct and extend distributed software systems through the use of object technologies. Object migration offers a way to control the fine-grained distribution of data, functionality and computations throughout networked computer systems.

Migration technology is used by a whole spectrum of software systems for very different kinds of objects. Several languages and their runtime environments have been extended or built specifically to support migration of atomic language objects as for example Emerald [Jul1993] or Trellis/DOWL [Ach1993b].

Some large software systems use migration as a means for reconfiguration of compound objects that are often also multithreaded like Eden [Bla1985a] or SOS [SG+1989]. In a broad sense, the term object migration may also be used for the migration of processes at the operating system level as processes encapsulate state and behavior and can be perceived as objects as in DEMOS/MP [Mil1987].

The spectrum of migration technologies can be interpreted as a continuum of design choices. Many fundamental techniques are applied across various forms of migration despite differences in the granularity of objects and the scope of the particular migration mechanisms. On the other hand, some problems of migration are common across all forms of object migration, the most fundamental one being the heterogeneity of the systems involved.

Heterogeneous Migration

Most approaches to migration assume that all participating hard- and software systems are equivalent or in other words homogeneous. Heterogeneity of systems is frequently defined only in terms of hardware differences, but various forms of heterogeneity can be identified at different levels of abstraction, especially within distributed systems. From differences of language environments to fundamental operating system and hardware platforms, distributed systems are almost by definition heterogeneous.

Heterogeneity has until recently not been a main focus of research in object migration. Most solutions offered so far effectively establish homogeneity within a heterogeneous system prior to migration through either virtual machine implementations or embedded interpretive environments as a prerequisite. The few approaches that differ address only a single form of heterogeneity [Shu1990, StJ1995] in most cases heterogeneous hardware through a conversion of object representations.

Apart from distinctions of hardware and operating systems as well as programming languages differences of libraries and applications do also have to be considered in the context of heterogeneous migration. Especially differences at the application level become more and more important in an increasingly interconnected world.

Object technology has enabled the transition of software construction from monolithic approaches to the combination of prefabricated parts. This transition has opened up a whole new dimension of heterogeneity among software systems at the library level that needs to be addressed by modern migration mechanisms.

At the same time end user scripting capabilities are used to extend existing applications with new functionality previously not envisioned by the original designers. Such programmable applications need to be treated in the same way as programming environments in order to make migration of user defined objects happen.

---

[1] The term "object migration" is also used for the reclassifications of objects within class-hierarchies, especially in object-oriented databases [MMW1994, Su1991, ToP1995].

Previous and current research efforts focus on specific details of the problem of heterogeneous migration. These efforts fail, due to their differing approaches, to conquer the overall problem in a consistent and extensible manner. Heterogeneous migration among existing language environments has not been addressed by any of the approaches investigated.

Related Technologies

Two software systems that are related to heterogeneous migration have drawn a lot of attention in recent years: the “Internet” programming language Java and the interoperability standard Common Object Request Broker Architecture (CORBA). Both systems do not implement heterogeneous object migration themselves but are used as foundations for various approaches to object migration.

Java

The programming language Java [GJS1996] offers a virtual machine environment for the execution of object-oriented programs that can be transferred over the Internet. The use of a virtual machine for the platform-independent implementation of object-based systems has been pioneered by Smalltalk [GoR1983] and is applied in this case to the heterogeneous Internet.

The virtual machine approach addresses the problem of heterogeneity through the creation of an embedded homogeneous environment within each participating heterogeneous system. It is therefore limited in its ability to make use of platform specific functionality as the various levels of heterogeneity are only addressed at the time of implementation of the virtual machine for a particular platform.

The Java programming environment itself is not an object migration mechanism as only object definitions are transferred over the Internet not the state or the computations of objects. Java implements a mechanism for the dynamic loading and binding of virtual machine code at runtime, which can be used to implement object migration as in Jada [CiR1997] or Mole [SBH1996].

Recently provisions for the transfer of externalized state of objects have been added to the Java specification in the form of an object serialization technique [Sun1999]. A full-featured object migration facility is nevertheless missing from Java, which is also not designed to allow the transfer of computations objects at runtime, but has to be modified significantly to do so as in Sumatra [ARS1996].

CORBA

The Object Management Group (OMG), a standards committee of commercial software vendors has released several specifications relevant to migration in so far as conformant software products are used by a number of migration systems. Although not a migration technique in its own right these cross-platform nature of these standards is helpful to address problems of heterogeneous migration.

The Common Object Request Broker Architecture (CORBA) [Sie1996] defined by the standardization organization Object Management Group (OMG) offers remote method invocation across programming language, operating system and network boundaries. While CORBA generalizes the concept of remote method invocation across heterogeneous language environments it offers only distributed message passing not object migration.

In a more recent effort, the call-by-reference scheme of the CORBA remote method invocation mechanism is extended to call-by-value semantics. The specification assumes that compatible implementations exist for the transferred objects as the call-by-value semantics establishes them as copies within the destination environments.

Despite various approaches to heterogeneous object migration and the existence of several technologies that can be used to address different levels of heterogeneity, there is still no all-encompassing mechanism for the migration of objects among heterogeneous environments. Beyond that, there is not even a consistent level of understanding among researches about the issues and problems raised by heterogeneous object migration especially at the language level.

1.1 Research Goals

The main goal of this work is the design of a migration mechanism able to migrate objects including their semantics among heterogeneous environments at runtime, independent of differences of hardware, operating systems, languages, libraries and applications. The mechanism is intended as an application extension rather than, as a change to the definition of existing programming languages or to the implementation of programming environments. This work will therefore focus mainly on migration at the language level.

A secondary goal of this work is the determination of the issues and problems of heterogeneous object migration as well as of the scope of heterogeneous object migration among existing language environments and of the migration techniques necessary to extend the scope. A third goal of this work is the identification of possible applications of heterogeneous object migration at the language level.

1.2 Motivation

The most obvious reasons for research in heterogeneous migration are roughly the same as with migration in general as for example load balancing and resource allocation issues as well as questions of availability and fault tolerance [Smi1988, Ple1996]. Additional reasons are the support for mobile systems as well as for specialized hardware.

The main motivation for heterogeneous migration at the language level is the transfer of functionality between systems that will lead to new areas of applicability as for example distributed software deployment and maintenance as well as remote diagnostics. The following incomplete list illustrates the various motivations as is ordered roughly from most specific for heterogeneous migration at the language level to least specific.

Availability of Functionality

A less obvious reason for migration in general but of special interest to heterogeneous migration at the language level is the migration of functionality between software environments. A migration can - to a certain extent - enlarge the destination environment with additional functionality, for example through the transfer of software libraries that are required by the migrated entity.

Functionality that is transferred through migration can be considered as a reason for migration in its own right. The functionality that is offered by the objects to be migrated can provide enough value from the standpoint of the destination that a heterogeneous migration attempt will even be successful if some of the fidelity of the migrated objects is lost in the process.

Migration can as well be initiated deliberately to add functionality to the destination environment [BeP1998]. Heterogeneous migration can thus be used to offer functionality across system boundaries. However, the applicability of the migrated functionality may be limited as only the migrated entities will be able to use it unless the destination environment or one of its applications is prepared to make use of newly migrated functionality.

Heterogeneous migration at the language level is therefore most beneficial in the context of continuously evolving software systems where the work of “porting” individual functionality is less rewarding than its creation or is even outpaced by the overall rate of innovation. Collaborative environments where the shared work is constantly evolving may serve as the most illustrative example in this regard.

Maintenance

The maintenance of large distributed systems usually requires frequent software updates. The functionality of objects that are available at remote nodes can be updated through substitution with migrated objects of additional functionality. Even in the case of objects in operation, migration can still be beneficial as the added functionality can be used by objects that are created later in the process.

Object with additional functionality that have been migrated to a remote site for maintenance purposes will thus gradually replace existing objects if newly created objects use the most recently added functionality. A complete replacement of existing objects with new functionality

will require some form of substitution operation or reinitialization of the object system under maintenance.

If migration is applied as a maintenance measure, the participating systems employ at least heterogeneous application software, as maintenance would otherwise not be necessary. Heterogeneous migration especially at the language level may simplify maintenance in some cases as common functionality may only have to be developed and tested once while being able to be deployed onto many systems. The determination of errors on one system may also lower the effort to debug multiple existing implementations for certain kinds of failures.

Remote Diagnostics

As distributed systems become increasingly complex the debugging of distribution related problems can become tantamount if a total serialization of events can not be achieved. Static debugging techniques that allow the analysis of the state of systems at specific points of execution are no longer sufficient in all cases. Failures of distributed systems that are hard to identify tend to be transient and communication related.

The diagnosis of complex dependencies between sequences of events can be accomplished with the help of analysis programs that react to certain conditions. In order to perform their task appropriate diagnostic objects need to be placed and executed at specific remote locations [RA+1998]. Heterogeneous migration can be used to distribute such diagnostic objects to remote sites and migrate them between different nodes as necessary. Heterogeneous migration at the language level will simplify remote diagnostics as diagnostics objects do only have to be implemented and tested in one environment.

Mobile Systems

In the context of mobile systems, migration can be used to transfer objects between stationary machines and mobile systems, which become uncoupled from the connecting network [BaM1995]. Objects that for example analyze data can be moved deliberately from mobile to stationary systems before the mobile devices become disconnected. The migrated objects continue operation and yield results independently of their mobile source. As soon as a connection is reestablished, the results can be collected and presented by the mobile device.

In the case of mobile systems connected through wireless networks, migration also lowers communication costs. Repeated retransmission due to the varying quality of service is avoided by migrating objects onto stationary machines that access network services intensively. As mobile systems and stationary systems tend to differ in hardware, operating system and language environments heterogeneous migration can be particularly useful.

Specialized Hardware

Specialized hardware as for example floating-point processors or vector-processing units can be a motivation to migrate objects to a different environment [Ple1996]. In some cases, migration may simply be beneficial because complex mathematical tasks that can only be solved with computation intensive software algorithms can be sped up significantly though the utilization of hardware support on a different machine.

As using the additional hardware support of another machine is implies heterogeneity a corresponding migration will only be possible using heterogeneous migration techniques that are able to take advantage of the additional different hardware facilities. However, this area of applicability of heterogeneous migration will be confined to very special cases.

Fault Tolerance

Related to the general resource allocation problem is the question of availability and fault tolerance [RB+1998]. Communication links tend to be more unreliable than the computational nodes, especially in wide-area or even wireless networks. Migration according to availability deliberations can be particularly useful for high-reliability systems. Distributed systems that are supposed to be in continuous operation can be kept running through migration even if some of their nodes are likely to become unreachable or have to be shut down due to maintenance.

Migration can also be used as an adaptability mechanism for fault-tolerant systems if for example software entities are moved according to failure-probabilities. Heterogeneous migration widens the set of systems that can be considered for availability. Support for heterogeneity can also be beneficial for fault tolerance as weak points of one system may be compensated through migration to other platforms.

Security

Migration technologies can also be used to achieve a higher level of security. Local execution of code can provide better security control because sensitive data does not need to be transferred over insecure telecommunication links. In addition, access- and control-provisions are not compromised by varying protocol-functionality or access rights.

Heterogeneous migration allows movement of sensitive operations to specific machines that for example enforce special audit-functionality. The overall cost of secure systems can be lowered, as not all machines need to enforce the same high level of security. On the other hand the techniques necessary to support heterogeneous migration especially at the language level raise additional security concerns because the functionality that is transferred between systems needs to be trustworthy and the object to be migrated needs to trust its new host [Che1998, Vit1996].

Load Balancing and Resource Allocation

Last not least, load balancing is generally assumed to be the main purpose for the migration of processes. Whether idle machines are used for computation intensive tasks as in Locus [PoW1985] and Sprite [DoO1990] or whether the overall utilization of resource offered by a network of machines is optimized [Ste1998m], migration of processes allows systems to adapt to changes of computational load at runtime. In general, migration can be used in many kinds of resource allocation problems beyond load balancing.

In the case of intensive storage access patterns or higher than usual network bandwidth requirements migration can be used to choose the most suitable combination of machines at runtime or to minimize latency of access as well as communication costs. Once again, heterogeneous migration widens the set of nodes involved. On the other hand, the additional overhead for heterogeneous migration may influence resource allocation decisions and migration policies.

1.3 Related Work

Only few existing migration systems address issues of heterogeneity [ChC1991, Nut1994b, Smi1988]. Of those that do almost all consider only heterogeneous hardware not software environments. For example, no process migration system could be found that attempts to implement process migration between different operating systems.

Large software systems that use migration also tend to define their own homogeneous execution context in terms of libraries that implement transparencies. Support for heterogeneity is limited to the generation of code for each platform. Language based migration environments, although apparently better suited to address heterogeneity, emphasize compiler techniques and only few systems address aspects of cross language migration.

The following overview provides an incomprehensive lists of typical as well as unusual examples of migration systems and illustrates the spectrum of heterogeneous migration mechanism that exists today: The examples have been chosen for their relevance to heterogeneous migration at the language level and appear in alphabetical order of the corresponding authors.

> Bennett [Ben1990] discusses the management of class hierarchies across migrations in an implementation of distributed Smalltalk. Constraints towards the design of classes of migrateable objects are described and some alternatives are discussed.
>
> Cardelli [Car1995a] defines Obliq, a language with objects and procedures that uses closures to migrate only computations of objects among heterogeneous

environments using an intermediate representation format. Closures are also used by other systems like Dreme [Fuc1995] for heterogeneous thread migration.

Kono et al. [KKM1996] let computations migrate to an appropriate host in a network of heterogeneous systems in TrapDo. Passive objects that are referenced by computations are migrated to the particular node of execution through conversion of their data and references.

Lucco et al. [LSW1995] describe Omniware a commercial product that implements migration of complete application written in different languages. The approach requires the compilation of source into virtual machine instructions that are dynamically translated into native machine code at load-time.

Pleier [Ple1996] has developed a compiler technique that allows the transfer of process state between heterogeneous hardware platforms based on specialized executables for a pair of machine architectures. At defined migration points during the execution of the program code migration can be initiated. The necessary conversion of data and execution information is performed during migration.

Shub [Shu1990] extends the migration mechanism of the V-System [Che1988] with support for heterogeneous hardware. Based on a common memory layout used for the generation of code for each supported platform the contents of the address-space of the migrated process is converted in accordance with the destination.

Steensgaard and Jul [StJ1995] have extended the well-known Emerald system towards migration among heterogeneous hardware. Their approach focuses on compiler support for the generation of reentrant code for each supported platform and on migration-time conversion of object state and computations. Most aspects of migration within Emerald are upheld in the context of heterogeneity.

Theimer and Hayes [ThH1991] propose a source code level technique for heterogeneous migration of processes. For restricted procedural programs, startup-procedures are generated at migration time that recreate the state of the computation on the destination host. The technique is used in similar form by a number of systems like Dome [AB+1995].

Another kind of software environments that are related to heterogeneous migration are mobile agent systems. The so-called mobile agents are software entities that can move autonomously within a network of appropriate execution environments. Depending on their particular approach mobile agent systems both make use of existing migration techniques or implement migration techniques themselves.

Mobile agents that are implemented using object technologies are constructed from several atomic objects that interact to implement the behavior of an agent. Only few approaches implement agents directly as atomic objects in which case agent mobility becomes equivalent with object migration [Gla1998].

Many of the overwhelming number of mobile agents systems [CPV1997, GhV1997, Kna1996, Lan1998] that have been proposed and implemented in recent years rely on the Java programming language mentioned above to address problems of heterogeneity. Only few agent systems address problems of heterogeneity directly using different techniques.

Acharya et al. [ARS1996] describe Sumatra, an extension of the Java programming language environment that adds transfer of state and computations to achieve object migration capabilities. The approach uses a modified version of the virtual machine implementation of Java to address the problem of object migration and of heterogeneous hardware and operating systems at the same time.

Hurst et al. [HRS1997] use state machines to capture and transfer computations of smart messages in the Tourist Network (TNET) agent system. Using a semantic model based on state machines that is different from the usual programming source code TNET is able to abstract from differences of heterogeneous systems.

Kato et al. [KM+2000] have implemented the agent system Planet that uses a canonical representation to migrate objects using memory mapping via a persistent store between heterogeneous execution environments. The native representation of agents within a particular environment is converted to and from the canonical format used for the transfer.

Peine and Stoplmann [PeS1997] describe Agents for Remote Action (ARA) an agent system that employs interpretation based language implementations to achieve mobility of agents across heterogeneous systems. Using the interpretive approach Ara is able to transfer the state and computations of agents between appropriate heterogeneous execution environments.

The recently accepted Mobile Agent System Interoperability Facility (MASIF) Standard of the Object Management Group [ChC1997, MB+1998] defines common facilities for agent environments to transfer of agents and to interact which each other. However, MASIF assumes that a homogeneous language is used among agents systems, preferably Java, and does not support the transfer of computations.

Among all the migration systems and related environments that have been investigated throughout the course of this work, various aspects of heterogeneous migration have been covered. All approaches either extend and change existing environments heavily or invent completely new languages or software systems to implement heterogeneous migration. Interestingly enough, no mechanism could be found that addresses all levels of heterogeneity or migration among existing heterogeneous language environments.

1.4 Outline

The remainder of this work is composed of four chapters. The second chapter tries to provide an overview of object migration by systematically characterizing existing migration systems. This effort compares the various existing approaches in a coherent context and offers insights into the relation of the used techniques. Based on this characterization the problems of heterogeneous migration and the possible approaches towards their solution are discussed.

The third chapter describes a pragmatic approach towards heterogeneous migration at the language level that offers migration of object semantics among existing languages without changes to the objects or environments involved. The objectives of this approach are stated and the required preconditions are defined. All aspects of the mechanism, including a reference architecture, the necessary abstractions and the actual migration algorithm are presented. The feasibility of the mechanism is shown in a prototypical implementation. The design of applications for the mechanism and tools for their development are described. The use of the mechanism in a real world scenario is demonstrated in a detailed example.

In the fourth chapter, the migration mechanism is analyzed in the context of the systematic characterization developed in the second chapter for a discussion of its limitations. Simple enhancements to the mechanism are described as well as complex extensions that overcome limitations implied by the objectives. The applicability of heterogeneous object migration in the context of existing languages is discussed as well.

The fifth chapter provides examples of possible applications of heterogeneous object migration at the language level. The potential benefits for each particular case are explained and some necessary implementation details are discussed. A summary of the research results and a perspective of promising future research directions conclude this work.

1.5 Terminology

A new terminology is used throughout this work in order to avoid confusion that is caused by words that have different meaning in the context of the various programming languages involved. Language specific terms are used only in contexts that discuss aspects of particular languages or sets of languages. The new terminology is intended to be language-neutral.

The term *object* is used for all software entities that encapsulate state and behavior regardless of their creation and their relation to other software entities. The term *interface*[2] is used as a neutral term for the definitions of objects, i.e. of their state and behavior. These interfaces may be represented as source files or as objects themselves, e.g. as classes or prototypes, by a particular language environment.

The term *component*[3] refers to any part of the state of objects independent from the nomenclature of a particular language where a component may be named a variable, a member, a slot, etcetera. The term *signature* is applied to any part of the behavior of objects that is available to other objects, regardless whether it is named a method, an accessor function, a generic function or something else by a particular language.

1.6 Notations

Throughout this work the *italic* font-attribute is used to indicate terms that are described for the first time but will be used for the rest of the text. The fixed width font `Courier` is used for all source code excerpts that appear separately as well as for fragments like single keywords or names of interfaces that are appear in the normal text.

Embedded figures and tables are used to illustrate various aspects of object migration in detail. A common graphical notation is employed uniformly among figures in order to enable comparisons of different situations. If not specified otherwise all figures show the situation after migration has occurred.

Ellipses are used to represent objects that are named and numbered. Dashed bold ellipses denote the object to be migrated prior to migration and bold ellipses symbolize the migrated objects after migration. A double lined arrow is used to indicate the migration itself. Relationships among objects are shown as dashed arrows lines indicating relationships prior to migration and uninterrupted arrows after migration. Figure 1.a shows an example.

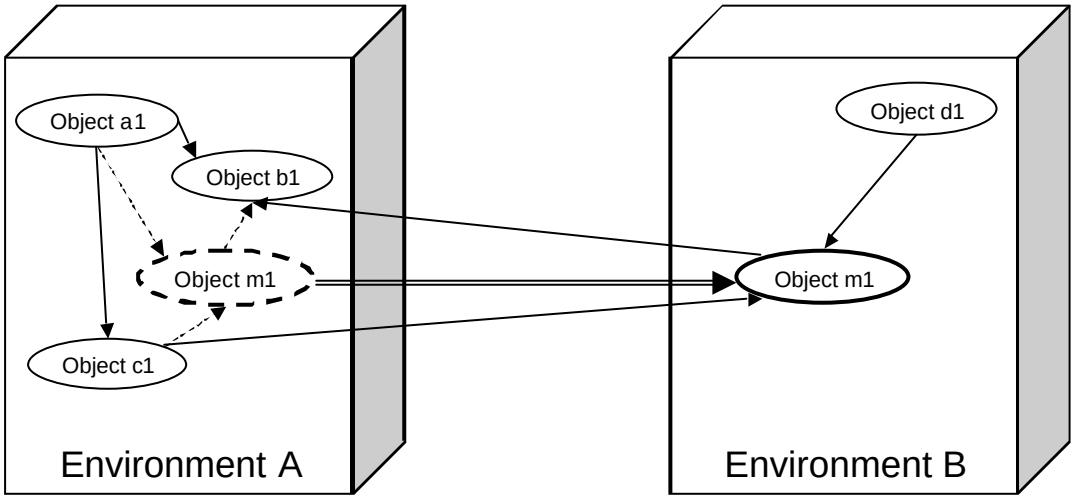

Figure 1.a: An example of a graphical notation used to illustrate object migration. All objects are shown here as ellipses. Object m1 has been migrated from Environment A to Environment B. The relationship between Object a1 and Object m has been cut. In contrast the relations between Object c1 and Object m1 as well as m1 and b1 have been substituted with remote references to and from the migrated object m1 in Environment B. The relationship between Object d1 and Object m has been newly established.

The distinction between objects and interfaces is visualized through the use of ellipses for objects and boxes for interfaces. Capital letters are used to name interfaces and small caps are used for object names. Unless mentioned otherwise subtype or inheritance relationships are

---

[2] The term “interface” should not be confused with the keyword `interface` of the Java programming language [GJS1996].

[3] The use of the term *component* for the state of objects should not be confused with the term software component that is often used to describe coarse grained software entities that are based on sets of atomic objects [BA+1995].

depicted from bottom to top. The “created by” relationship is displayed from right to left. Due to their implicit nature, dotted arrows are used for both relationships. If necessary, explicit relationships between objects will be shown as normal arrows. Figure 1.b shows an example of object definitions and related objects.

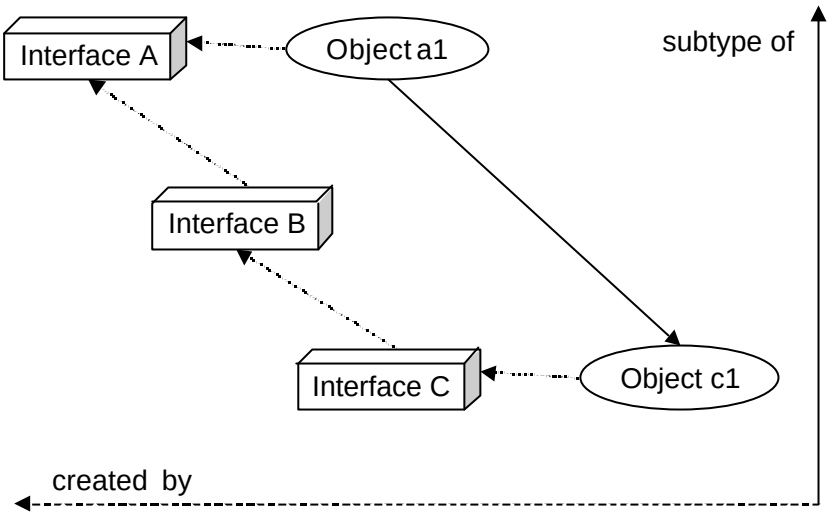

Figure 1.b: An example of the graphical notation used to illustrate the relations between objects shown here as ellipses and interfaces shown here as boxes. Interface C is a subtype of Interface B, which is in turn a subtype of Interface A. Object a1, was created from Interface A and references Object c1 which was created from interface C.

Tables provide comparisons of general concepts as well as overviews of the relationships between concepts and characteristics. Wherever possible, tables share the same structure. General concepts are listed at the top and the particular characteristics currently discussed are shown in the leftmost column of the table.

Simple crosses "x" indicate the presence of a particular relationship. Parentheses around crosses "(x)" are used when exceptions apply. Empty parentheses "()" denote that a particular characteristic cannot be applied in general because only one or a few examples of the respective concept with that characteristic exist. Table 1.a provides an example.

	compiled	**interpreted**
Binary	x	
Virtual		(x)
Intermediate		(x)
Source		()

Table 1.a: An example of a table showing the relationships between various concepts and characteristics. All compiled languages[4] use a binary representation, as for example, C++ [Str1991]. Interpreted languages, either use virtual machine code as Java [GJS1996] or Smalltalk [GoR1983] or intermediate representations as Obliq [Car1994] indicated here by parentheses around an "x". Only few languages interpret source code directly as indicated by empty parentheses.

Source code excerpts, figures and tables are numbered separately for each chapter using lower case letters in alphabetical order. References to other parts of the text include the number of the page as well as the chapter unless the page is contained in the same chapter as the reference itself.

[4] The exception of virtual byte coded compiled to binary machine language that is offered for example by some Java implementation is not covered here in order to avoid confusion.

2 From Process Migration to Heterogeneous Object Migration

The term *migration* denotes the transfer of software entities between information systems at runtime. The kind of systems that support migration, the entities that can be migrated and the characteristics maintained through migration differ among migration systems but all approaches share some common aspects.

Unlike other distribution techniques as for example configuration management, migration is initiated and performed at runtime. The source and the destination environment as well as the software entity to migrate do not have to be predefined but can be determined dynamically. The migrated entity is transferred to the destination environment completely[5] and is able to function at the destination in the same way as within the source environment.

Existing approaches to migration can be roughly subdivided into migration of whole processes of operating systems, migration of complex compound objects of software systems and migration of fine-grained objects of object-based programming languages. The kind of software system that implements migration is in most cases correlated to the granularity of the software entities that can be migrated.

The software entity that can be migrated determines not only the amount of information that is transferred between systems. It also defines the requirements of the particular migration mechanism, the migration techniques that can be applied and the kind of decisions that have to be made during the initiation and control of the migration operation.

The actual algorithms that are used to perform migrations are implemented as operating system services, as special layers of software systems or as parts of the runtime environments of programming languages respectively. The terms *process migration*, *system migration* and *language migration* will be used to refer to the corresponding levels of granularity of migration.

Heterogeneous environments are only poorly supported by existing migration systems, but various opportunities of extension towards heterogeneity can be determined. Unless mentioned otherwise the following overview of characteristics of migration systems will be confined to homogenous cases where identical source and destination environments are assumed. Heterogeneity will be dealt with in detail at the end of this chapter (see page 57).

Since many aspects of migration systems are interrelated, no single outline hierarchy can bring all aspects into a linear order. As a consequence, some repetitions are inevitable and progressive disclosure is used to bring all issues into a logical order. Terms and concepts not previously mentioned are described briefly at their first appearance in the text and forward references are used to hint at more detailed discussions of the respective topics.

[5] Unlike replication, objects are not copied but moved through migration.

2.1 Unit of Migration

The most general and obvious characterization of migration systems derives from the question "What software entity is migrated?". The term *unit of migration* defines the smallest software entity that can be transferred between systems. Many migration systems are able to migrate several units of migration simultaneously [Sch1990] but only the smallest migrateable entities of each case is referred to here. A unit of migration can be a process of the operating system, a "coarse-grained" object of a software system or a "fine-grained" object of an object-based programming language.

- language objects - language migration

  First class objects of programming languages define one end of the migration spectrum as they are atomic and cannot be subdivided into smaller parts[6]. Fine-grained objects of object-oriented programming languages are called *language objects* and the term *language migration* will be used to denote their transfer between environments.

- system objects - system migration

  Object-oriented software systems often define more complex "objects" as independent entities, which usually consist of occasionally large amounts of fine-grained objects. As they are not subdivided for migration but transferred as a whole these compound objects will be called *system objects*, and their migration will be referred to as *system migration*.

- process objects - process migration

  Processes are characterized by the address space they encompass and by the process context that the corresponding operating system maintains. Since the term *process migration* is already well established and many techniques of process migration are similar to system and language migration, processes will be called *process objects* in order to provide a consistent terminology.

Process migration is subsumed under object migration as one extreme end of a spectrum with the migration of atomic language objects as the other extreme and system migration as a middle ground. The use of the term object for processes is not new as it has been used for various operating systems entities since the pioneering work of Hydra [WC+1974]. Figure 2.a illustrates the differences between the three major units of migration.

The subsumption of all "units of migration" under the term "objects" is guided by the fact that the migrated software entity at least logically encapsulates all the information necessary for the entity to be able to work as intended within the destination environment. With the exception of language objects, the term "object" does only provide a loose characterization of the software entities that can be migrated.

Language objects are usually created on the basis of formal object definitions, that define the structure of the state and the semantics of the behavior of objects. These object definitions are called *interfaces*. Some systems like Smalltalk [GoR1983] or Self [Sun1992] represent these object definitions as objects in their own right while other systems like C++ [Str1991] manage them as source files. System and process objects also contain state and show behavior but their semantics in terms of executable operations are usually defined only via source files.

Mobile agent systems employ similar technologies than object migration but are less clearly distinguished as *mobile agents* are implemented on all three levels of granularity, depending on the particular approach. For example, TACOMA [Joh1998] implements agents of the size of processes, while Mobile Object Workbench [BH+1998] employs system objects composed from finer grained objects and Voyager [Gla1998] uses atomic language objects.

Each aspect of migration described in the following subchapters will also be discussed with regard to the three basic units of migration. Clearly marked sections are used to separate the

[6] The distinction between objects and values often found in hybrid object-oriented programming languages is not considered here, as values are in all known cases not migrateable by themselves but only as part of the externalized state of objects.

particular details from more general discussions. Existing migration systems including mobile agent systems will be cited as illustrative examples wherever appropriate.

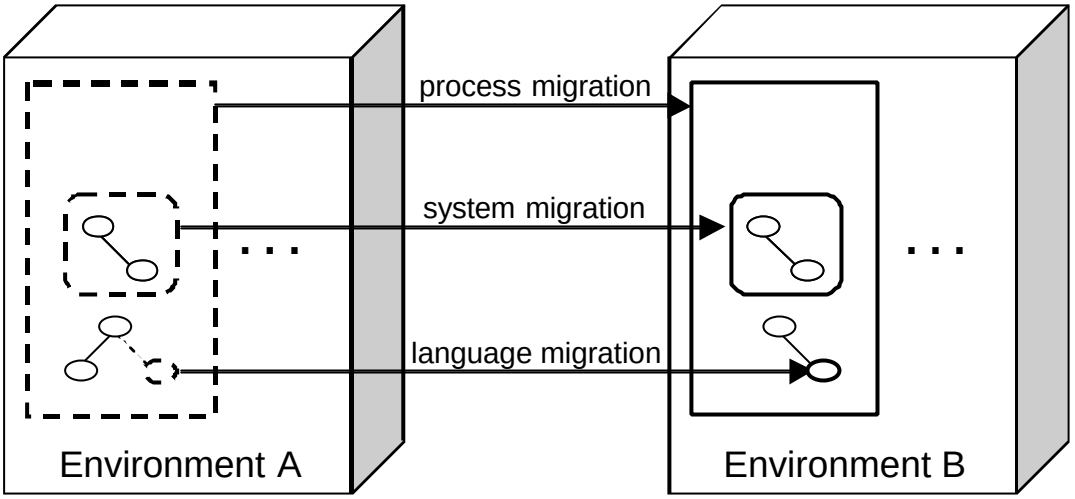

Figure 2.a: The "unit of migration" that is transferred between environments may either be a whole process object (shown here as a rectangle), a complex system object (shown here as a rounded rectangle) that is composed from finer objects, or a single atomic language object (shown here as an ellipses). All three kinds of migration and the implicit inclusion among objects are shown here collectively only for illustrative purposes.

For each "unit of migration" four kinds of information characterize an object and need to be transferred during migration: identity, state, functionality and computations. These four kinds of information are described here briefly and will be discussed in more detail in the following subchapters.

- identity

  The *identity* of an object characterizes its uniqueness with regard to a context, for example among all other objects of a language environment. The identity of an object is usually maintained throughout its lifetime. In the context of distribution, the identity of an object can be extended to distinguish it among all other objects of a distributed system. The transfer of the identity of an object implies that there is only one object of this kind at any given time within the context of the distributed system including during and after migration.

- state

  The *state* of an object encompasses all data that is stored within the object as well as all references to other local and remote objects. References to other objects imply dependencies between objects. The transfer of the state of an object maintains all the data that is necessary for the object to operate as intended by the developer. This may or may not include references to other objects and may lead to the transfer of related objects as well. The transfer of state is therefore related to the notion of identity.

- functionality

  The *functionality* of an object, also called its *behavior* or *semantics*, consists of all operations, usually called *methods*, the object is able to perform. The transfer of the functionality of an object involves an appropriate representation of all operations that are necessary for the object to operate within the destination environment.

- computations

  The *computations* of an object subsume all operations that are currently being carried out by the object at the time of migration. Operations the object takes part in, for example as a parameter, may imply additional dependencies between objects. The transfer of

computations of an object involves all the information that is necessary to continue the current activity of the object at the destination.

Not all of these four kinds of information are transferred by all migration systems. The question "what is encapsulated by an object as its integral part and what is assumed to be part of the environment" determines the amount of information that is transferred at migration time. The amount of information transferred differs with the unit of migration as well as with the level at which migration is performed.

A process object usually encompasses all four kinds of information while less information may be transferred for system and language objects. Identity and state is transferred by most migration systems, but functionality is only transferred if necessary and only few systems support the transfer of computations. Table 2.a provides an overview of the various kinds of information transferred with regard to the unit of migration.

	Process migration	**System migration**	**Language migration**
Identity	x	x	(x)
State	x	x	(x)
Functionality	x	(x)	(x)
Computations	()	()	()

Table 2.a: The amount of information transferred differs for each kind of migration system. Empty parentheses indicate only few systems that transfer the particular kind of information and parentheses around "x" indicate only few exceptions from a general support of transfer of information which is indicated by a single “x”.

As table 2.a indicates, not all language migration systems transfer the identity of objects but alternatively establish copies of migrated objects [Sil1996]. Some language migration systems do not migrate state as Obliq [Car1994]. Many language and system migration mechanisms also rely on pre-distributed code and do not transfer functionality of objects during migration as for example DOME [AB+1995].

Only few language and system migration mechanisms migrate computations, some migrate so called *threads* independently of objects as TrapDo [KKM1996]. Even computations of the operating system kernel that are not encapsulated by address-spaces are in some cases transferred during process migration as in Accent [Zay1987a].

Examples of the transfer of each kind of information can be found for migration at all levels. Yet, only few systems at each level cover the whole range of information that can be transferred during migration. The following subchapters discuss each kind of information transferred in more detail.

2.1.1 Identity

The identity of an object is the one characteristic that distinguishes a particular object from all other objects with regard to a certain context. Objects are unlike values within conventional, e.g. procedural programming systems not distinguished by an equivalence relation of their content. Instead, they are characterized only by their identity which can be called their *local identity* in the context of a single environment.

In the context of distribution the identity of objects is related to the problem of finding, addressing and accessing objects on remote systems. This is usually achieved through remote references which contain some kind of location information. Symbolic names can be used additionally to establish remote references through a lookup of the location information via name servers (see also page 42). The identity of objects in the context of distribution can be called their *distributed identity*.

When several related objects are considered as a set of dependent objects, their identity can also be defined relative to each other, especially if distributed identity is not supported. The

topology of references between objects of a set of objects can be called their *relative identity* in the context of the particular set.

The notion of identity of an object may change in the context of migration. If supported by both source and destination environments the distributed identity of objects is transferred in most cases and relative identity is maintained implicitly . If distributed identity is not supported, relative identity needs to be maintained explicitly in the context of the sets of objects that is migrated and in some cases also for subsequent migrations [Sil1996]. The local identity of the objects being migrated will be lost in any case. Figure 2.b illustrates the various forms of identity in the context of object migration.

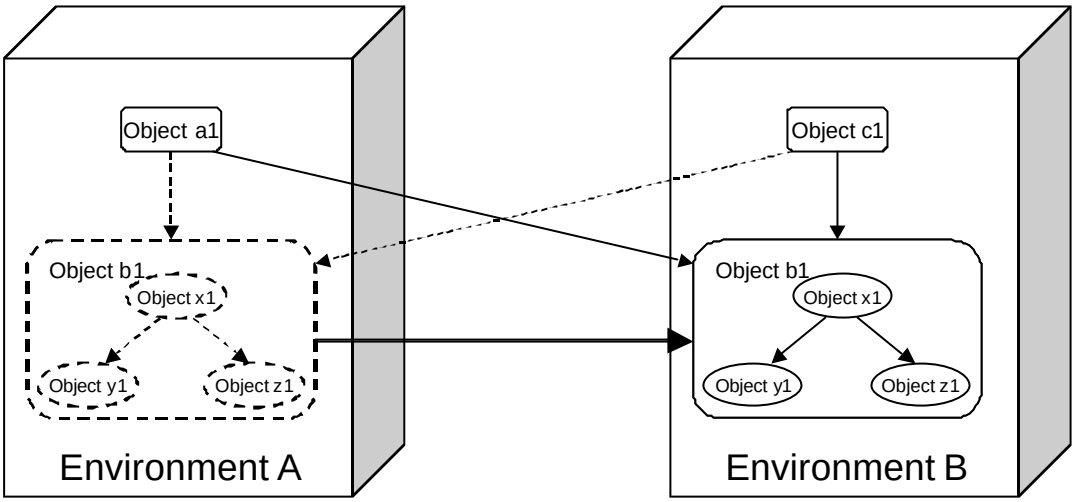

Figure 2.b: The difference between distributed and relative identity of objects can be exemplified by the migration of system objects (shown here as rounded rectangles) that consist of language objects (shown as ellipses). In the above example, system object a1 references system object b1 prior to migration through its local identity in the context of environment A. System object b1 consists of object x1, y1 and z1 which are related via their relative identity in the context of system object b1. System object c1 references system object b1 in a distributed context. After migration the relative identity of objects x1, y1 and z1 is maintained within the destination environment B, while system object a1 now references object b1 in the context of distribution and system object c1 now uses the new local identity of system object b1 within environment B.

Distributed identity can be maintained during migration if local references can be extended to include location information as in Emerald [Jul1993]. Such references can be called *implicit remote references.* If what can be called *explicit remote references* are distinguished from local references, distributed identity has to be substituted for local identity during migration.

If distributed identity cannot be established during migration, local references can only be cut and the participating objects have to be aware of that fact. Symbolic names are used by some systems as an alternative to remote references, but in most cases, especially in object-based systems proxy objects are used as in SOS [SG+1989].

Proxies are local objects that represent remote objects through encapsulation of the location information. They forward local messages to the remote object and return the remote results locally. Proxies substitute their local identity for the distributed identity of a remote object, effectively hiding the complexity of maintaining the distributed identity from the local environment and all local objects (see also page 42 for a detailed discussion).

Maintaining location information is a major problem for migration systems as Zayas states:

> "Part of the [migration] problem lies in providing an efficient method for naming resources that is completely independent of their location." [Zay1987a]

If an object is migrated its location changes and the location information of remote reference that refer to it will become outdated. The location information can be corrected using mechanisms like forward pointers or name servers as well as more costly operations like immediate update at the time of migration or broadcasts of queries at the time of access. These update mechanisms also differ in the way they cope with network failures and partitions (see also page 44 for a detailed discussion).

The identity of objects is tightly related to the state and computations of objects described in subsequent subchapters. The management of identity in the context of migration defines how remote objects can be accessed, how objects can communicate and how the flow of control between objects can be handled in the context of distribution and migration.

Process Migration

Operating systems distinguish processes usually through process identifiers that are defined locally. Distributed operating systems and especially those that support process migration, use globally unique process identifiers that represent distributed identity as for example DEMOS/MP [PoM1983], Locus [PoW1985], Sprite [DoO1990] and System V [Che1988].

Only few systems support the migration of process groups [Che1988], but the maintenance of relative identity does not appear as an issue in the presence of distributed identity. This is also true for systems that extend process groups across machines as for example Rhodos [PG+1996]. In general, a migrated process can be running under a different local process identifier on the destination machine if it can be addressed through an inter process communication mechanisms independently of its migration.

During migration, some systems create a process with a different distributed identity at the destination host as for example Accent [Zay1987a]. The purpose is to use normal memory transfer operations that require two distinguishable entities. When the migration commits, the new process at the destination is given the distributed identity of the source process which is subsequently destroyed.

System Migration

The distributed identity of system objects is defined within the design of the respective software system usually in terms of a message passing or persistence mechanisms [SiA1996]. As system objects often communicate via messages within a distributed environment, they can usually be addressed via symbolic names or proxy objects as in SOS [SG+1989].

Large software systems often employ a so called two level object model that consists of coarse grained system objects with distributed identity as well as of fine grained atomic objects with local identity as for example in COOL [AJ+1992], Eden [Bla1985a] or SOS [SG+1989]. The distributed identity of system objects is maintained during system migration as well as only the relative identity of the finer grained objects.

Language Migration

The local identity of objects within programming language environments is in most cases defined via memory addresses. These are used for local reference to objects as well as by the message passing mechanisms. Since local memory addresses are not suitable for the distinction of objects in distributed environments the notion of identity needs to be augmented.

Globally unique addresses of objects are used directly only within environments that have been designed with distribution in mind as for example Emerald [Jul1993]. If globally unique addresses are used the local, relative and distributed identity of objects are identical which simplifies many aspects of migration as for example the management of references between objects as well as the transfer of computations.

The most common way to handle the distribution of objects at the language level is the use of local proxy objects that are referenced and addressed by message passing just like any other local object as in DC++ [ScM1993b] or Brouhaha [DNX1992]. Proxy objects encapsulate remote references to distributed objects, forward messages to these objects and manage the remote access to parameters as well as the return of results.

Proxy objects maintain the local and relative identity of migrated objects while their distributed identity is hidden. A dependency between the distributed environments involved and the additional complexity for the management of proxy objects during migration are consequences of the use of this technique (see also chapter 4 page 159 for a detailed discussion).

2.1.2 State

The state of objects encompasses all information that is encapsulated and used by an object during its lifetime. An objects state needs to be represented appropriately for its transfer during migration, e.g. values of atomic objects have to be encoded in order to be transferred over the network. Several representation formats can be used for the transfer of state (see page 50).

References to other objects are usually part of the state of objects and extend the encapsulated state through dependencies on other objects. Further dependencies are implied by references of other objects to the object to be migrated. These dependencies have to be handled appropriately by the migration mechanisms, e.g. through the use of a so-called *externalization* of a whole set of objects as the representation for the transfer between environments.

If the references used within an environment are aware of distribution like in Emerald [Jul1993] they can be used for local and remote objects alike. Such references are not affected by the migration of objects except for the update of these references with new location information. If only local references are available, proxy objects can be established for the migrated objects in order to maintain the dependencies in the context of distribution (see also page 42).

Otherwise, the dependencies between the objects have to be considered prior to migration and need to be handled properly by the migration mechanism. Objects that are dependent on each other can either be migrated together as a set or references have to be cut deliberately prior to migration and the objects involved need to be ware of this destructive operation. The determination which objects have to be migrated together can either be done at development time or it can be performed dynamically at migration time.

Process Migration

During process migration the address-space that defines a process to be migrated is transferred to the destination machine in almost all cases in binary form (with the exception of [ThH1991], Dome [AB+1995] and related systems). No distinction is made between memory segments of the address space, i.e. code, data, heap, and stack areas. As address-spaces tend to be quite large, sophisticated migration techniques have been developed in order to minimize the required to transfer the state [RoC1996] (see also page 52).

The process context of the operating system is also transferred to the destination as part of the state of a process object. Depending on the operating system, the process context may consist of details like register contents and address-mappings. Some process migration systems support even the transfer of the state of input/output operations and/or inter process communications as Sprite [DoO1987] or Rhodos [PG+1996]. Other redirect I/O operations instead as System V [TLC1985].

If a distributed file system is available, local file operations at the source can be relayed as remote file operations from the destination. Direct access to local hardware, for example access to frame buffers or coprocessors, can never be migrated and processes that make use of such facilities are therefore considered immobile. Some mechanisms provide shadow processes on the source system to provide continued access to local hardware like Sprite [DoO1990].

References between processes do not create an issue for the transfer of state of processes if globally unique process identifiers are used. Dependencies between process objects do only seldom influence process migration as inter process communication mechanisms usually work in the context of distribution. As an exception, process groups are maintained across different nodes in Rhodos [PG+1996].

System Migration

The state of system objects is often transferred between environments in the form of their binary in-memory representation. Alternatively, an externalized representation format is used to

recreate the particular system object in the context of the destination environment. While the latter case is similar to the transfer of state for language objects (described in the next section), the memory transfer for system objects differs from the migration of processes.

Because system objects do in most cases not encompass whole address spaces (one notable exception being Eden [Alm1985]) it is unlikely that a system object may be able to occupy the exact same memory addresses within the destination address space it is transferred to. Hence, some form of memory mapping has to be performed during system migration either explicitly by the developer or implicitly by the software system as in TrapDo [KKM1996].

Explicit memory mapping involves the relocation of memory contents to new addresses and the update of *pointers* to memory addresses within the state of the transferred system object. This is a rather tedious and time-consuming process that also requires complete knowledge about the location of pointers within the memory contents to be mapped.

As a consequence some software systems that support system migration employ implicit memory mapping abstractions as system objects are often bound to memory segments that can be mapped and addressed independently as in COOL [AJ+1992]. The relocation to the new address range is then performed implicitly through the application of a segment offset at the time of memory access. Figure 2.c illustrates the relocation of memory segments during system migration.

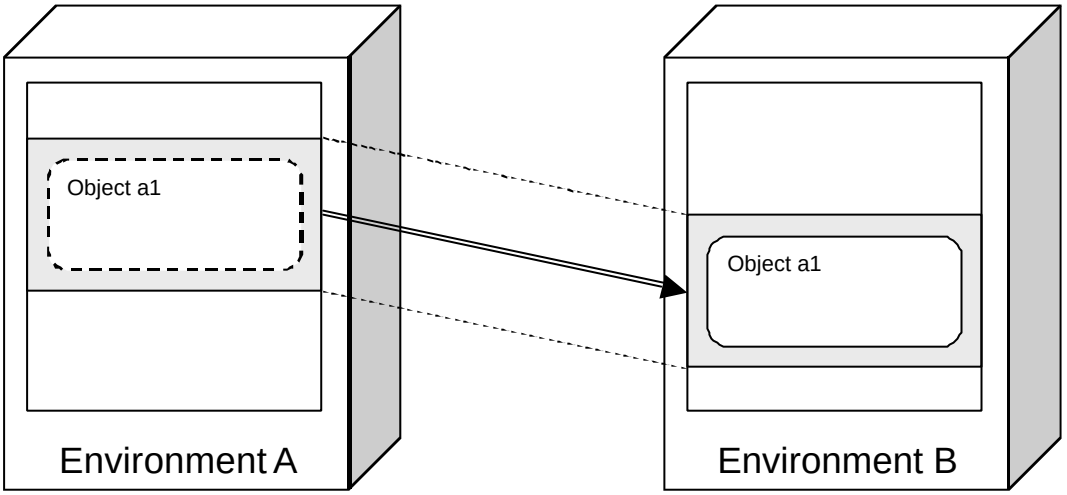

Figure 2.c: Address translation has to be performed during the migration of system objects that are shown here as rounded rectangles while address spaces are shown as rectangles. In the above example, the memory segment that represents object a1 is relocated within the destination environment to a new address.

A notable exception from memory mapping is provided by Amber [CA+1989b] that maintains a global virtual memory, that implements identical mapping of virtual memory addresses across all nodes. Objects to be migrated can be located at the exact same memory location on all nodes and references between objects remain consistent, leading to “object faults” if the referred objects have not been mapped locally.

System objects depend much more directly on other system objects than for example processes [Lux1995a]. Hence dependencies between objects have to be considered during migration of system objects and may lead to prerequisite, parallel or subsequent migrations of other system objects [Sch1990]. Clusters of large grained objects are for example the usual unit of migration in Cool-2 [JJC1995].

Agents systems also frequently use groups of objects to represent agents. As a consequence clusters of objects are transferred as a whole as in Mobile Object Workbench [BH+1998], Mole [SBH1996] and Sumatra [ARS1996]. A notable exception is Ara [PeS1997] which does not migrate the state that is considered external to an agent.

Language Migration

The state of a language object consists of references to other objects and in the case of hybrid languages also of values of primitive types. During migration references to other objects can either be maintained in the context of distribution, recreated at the destination through the additional migration of the related objects or deliberately cut.

The state of an object being migrated is transferred during language migration in a linearized form, in some cases including referenced objects recursively, and used to recreate the object within the destination environment. If supported by both environments remote references of the migrated object can be maintained by the migration mechanism.

If no other measures are taken local references will simply be cut by migration which can lead to disastrous consequences if it happens unintentionally and is not handled properly. Figure 2.d illustrates the different ways in which relationships between language objects can be treated during migration.

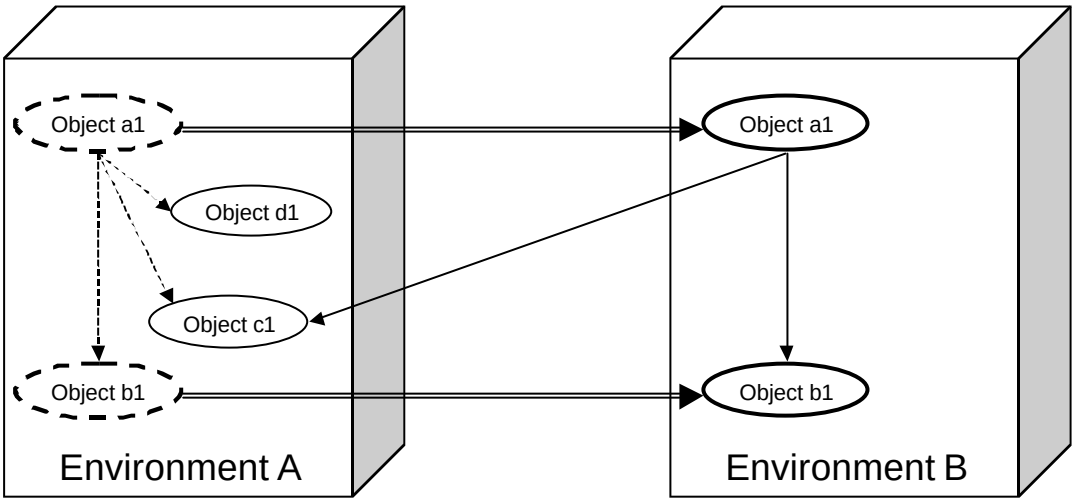

Figure 2.d: Relationships (shown here as arrows) between objects (shown as ellipses) can be handled differently during migration. In the above example the reference between object a1 and b1 is recreated at the destination as both objects are transferred, the reference from object a1 to c1 is maintained in the context of distribution and the reference from object a1 to d1 is cut.

The direct memory transfer is only rarely used [StJ1995] even in environments that support globally unique addresses because complex mappings of memory representations are involved. These mappings have to be performed explicitly as individual objects are usually too fine grained in order to apply relatively coarse-grained memory abstractions like memory segments. Complete knowledge about memory pointers within the memory representations of objects is also required in order to perform the necessary mappings consistently.

In some cases proxy objects are substituted for migrated objects in order to maintain references between objects. The set of objects that are actually migrated has to be defined at compile time for some environments as Emerald [Jul1993] while other enable dynamic decisions at runtime as Amber [CA+1989b].

2.1.3 Functionality

The functionality of an object encompasses all operations the object is able to carry out as specified by the object definition represented in-memory or in the corresponding source code file. The in-memory representations of operations involve actual machine code, virtual byte code or in some cases intermediate representations as in Obliq [Car1994].

The functionality of objects to be migrated is usually either dependent on other object definitions or on common functionality provided by language environments, software systems or operating systems respectively. The availability of this common functionality is in most cases a necessary requirement for migration.

The availability of the individual functionality of the objects to be migrated within the destination environment is also a prerequisite to migration but only seldom considered. Most systems assume that compiled code for each supported platform is already distributed among the participating environments prior to the actual migration.

System that do not assume the deployment of code prior to migration either transfer the necessary functionality in a separate phase of the migration process or as part of the object being migrated. In most cases, functionality of other objects already available at the destination is not taken into account during the migration process.

Another issue that has drawn a lot of attention especially in the context of agent systems is the security of the objects being transferred which is in most cases bound to the corresponding functionality [Vit1996]. Both the object being migrated as well as the destination environment may be subject to security threats through the transfer of functionality. The destination environment may be attacked by a migrated object while the object itself may as well be attacked by a malicious host.

A number of solutions have been provided most of which employ encryption capabilities [CG+1995] to either authenticate environments and objects in order to establish a trust relationship and to authorize mutual access between objects and environments on an individual basis. As this issue is only marginally related to migration and a lot of material on this issue can be found in the literature (Ajanta [TrK2000], Ara [PeS1997], Telescript [TaV1996]) this aspect will not be covered in greater detail here.

Process Migration

The functionality of a process is defined by the executable program that is used to create the process. It is encapsulated as machine code within the code segment of the corresponding address space. Process migration therefore transfers functionality as an inherent part of the address-space and does not need to treat functionality differently from state. The functionality transferred in binary form is dependent on the hardware architecture and on the operating system services.

Due to these dependencies only few attempts have been made to transfer processes between different hardware platforms and no approach to the migration of processes between different operating systems could be found. Theimer and Hayes [ThH1991] for example describe a process migration facility that overcomes differences of hardware platforms through recompilation and Arabe et al. [AB+1995] as well as Pleier [Ple1996] use precompiled platform dependent code for the same purpose.

System Migration

The functionality of a system object is usually dependent on the semantics of the underlying software system. System migration generally assumes that the common functionality of the software system is already available at the destination usually in the form of libraries for each supported platform.

The functionality of the system object to be migrated or its application is also often assumed to be available at the destination. This can be done either explicitly in the form of executables that are deployed with the installation of the corresponding application prior to migration, or implicitly through a central code repository or via a distributed file system that is accessed as necessary.

If the functionality of system objects is not already available at the destination, system migration mechanisms transfer the required functionality separately prior to the actual migration. This occurs in the form of libraries or executables that are dynamically bound within the destination environments. Only few approaches to system migration transfer functionality of system objects at the same time as identity and state like for example Birlix [Lux1995a].

Language Migration

The functionality of fine-grained objects is usually specified by object definitions. Depending on the programming language and its particular implementation, first class objects describing the functionality may be used as in Smalltalk [GoR1983] where a hierarchy of so-called *class*

objects is maintained in memory. Alternatively, the actual machine code may be directly referenced by the object itself as in C++ [Str1991] where object definitions are only represented as source files.

Again most language migration systems require functionality to be deployed prior to migration like DC++ [ScM1993b] and essential migrate only state [Sch1992d]. If the availability of the necessary functionality at the destination cannot be assumed prior to migration the functionality has to be transferred during migration as in EmeraldOS [StJ1995].

The transfer of functionality during migration has to be performed at runtime. And it requires some form of dynamic loading and binding by the destination environment because new virtual machine code or native machine instructions have to be made available. If a source code representation of object definitions is transferred an additional compilation phase will be necessary. In extreme cases, intermediate representations of functionality are converted to machine code during migration as in Omniware [LsW1995].

If dependencies between object definitions exist a set of multiple object definitions will have to be transferred. Figure 2.e illustrates the migration of an object definition as a side-effect of the migration of a language object.

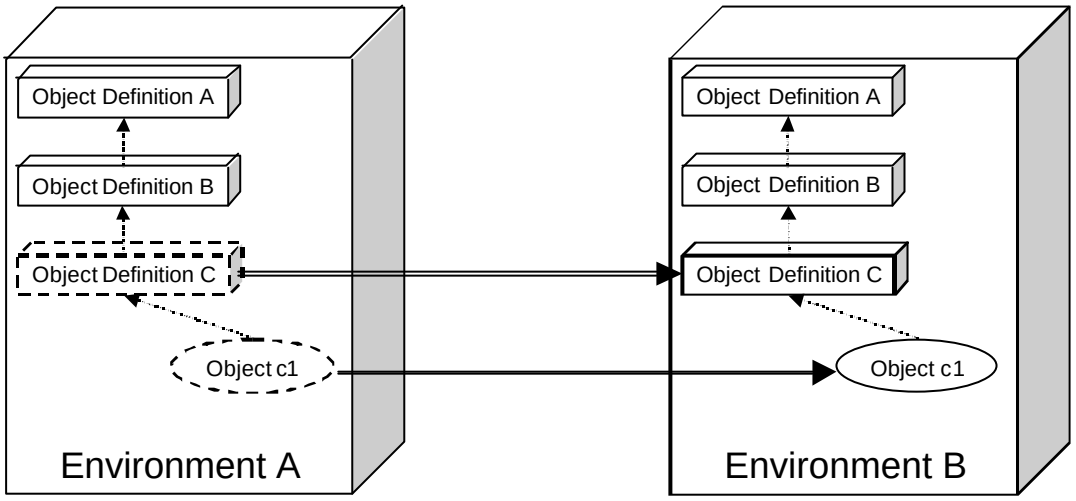

Figure 2.e: The definition of an object (shown here as a box) may be migrated along with the object (shown as an ellipses). In the above example, the definition C of object c1 is transferred to the destination environment B as a consequence of the migration of object c1.

The management of the distribution of functionality is only seldom considered with the notable exception of Bennett [Ben1990] who discusses several alternatives for an implementation of Distributed Smalltalk. HERON [FJ+1993] and JavaParty [PhZ1997] use the notion of distributed inheritance to manage the fact that objects and classes may be located on different nodes.

2.1.4 Computations

The computations of an object in the context of object migration are the active operations of an object that are under execution at the time of migration. These are represented as a stack of activation records. An *activation record* consists of the parameters of the operations, i.e. values and references that are passed as arguments at the invocation of the corresponding method.

The combination of a stack of activation records and an execution counter forms a *thread*, a term commonly used as an abstraction for a single flow of control within a program environment. The migration of computations separately from objects is also called *thread migration*. A system is called to be *multi-threaded* if several threads exist in parallel that are executed concurrently otherwise it is called to be *single-threaded*.

Computations can be called *internal* to or encapsulated by an object if only functionality that is part of the object itself is invoked. Otherwise, computations can be called *external* to an object. A process encapsulates all computations of the program it is created from with the exception of calls to operating system services that are external to the process.

External computations are the norm for system and language objects regardless whether their environment is single-threaded or multi-threaded. The only exceptions are active objects, i.e. system or language objects that encapsulate threads and interoperate with other objects only through explicit messages.

Regardless whether active objects are implemented single-threaded or multi-threaded, the threads encapsulated by active objects will only contain activation records of operations defined for the corresponding active object. I.e. threads encapsulated by active objects do not mix activation records of operations of different objects.

As a consequence, the migration of computations of active objects is simplified as all activation records of the encapsulated threads have to be transferred and only references to other objects have to be updated. The migration of computations of passive involves the determination of the activation records that have to be migrated (see also chapter 4 page 179).

Whether migration is initiated *preemptively* at any point in time or *non-preemptively* , i.e. only at specific points in the flow of execution also influences the amount of work that has to be done to transfer computations. In both cases, thread migration involves capturing and recreation of the computational state, i.e. the stack of activation records.

In the case of non-preemptive migration, the initiation of migration is performed by a normal method invocation and the state of computation is comprised of all previous activation records on the stack. If migration can be initiated preemptively for example through a software interrupt that can be serviced between any two machine instructions the state of computation will be much harder to capture.

In the case of a preemptive migration some processor registers may contain temporary values and the current activation record may not be up to date. Even worse sophisticated optimization techniques like code relocation and register allocation can further complicate the determination of the actual state of execution as well as its recreation within the destination environment [Ple1996]. Many systems therefore define so-called *migration points* within the code at which migration can be performed [SmH1996]. Preemptive migration request are delayed by these systems until the next migration point is reached through normal execution [Ach1993a].

Due to its complexity many migration systems do not transfer computations at all but require that objects are *inactive* to be migrated. The inactive status means that all methods that have been previously invoked for the object have already returned prior to the migration request. The migration of inactive objects is also called *weak migration* while the term *strong migration* includes the transfer of computations.

If threads are migrated independently of objects, several variants of thread migration exist. In the simplest case, a thread simply distributed when a remote method invocation is performed. A remote message invocation will transfer the flow of control to a remote system where the necessary functionality is already available. Additional activation records will be created within the remote environment until a result is returned together with the flow of control to the calling environment. The local computations will block until the remote invocation returns.

In this context asynchronous message passing can be viewed as thread migration or as a thread copy operation if a new thread is created within the destination environment for each incoming message. Alternatively, message queues can act as a distributed rendezvous mechanism for threads that reside on different machines and communicate only via messages.

A variant of the remote procedure call or remote invocation technique is *remote execution*, a technology that transfers computations in the form of *closures*, i.e. as a set of operations and a set of parameter that in combination are ready to execute. Remote execution resembles a remote procedure call that includes the code to be executed and the parameters as well, which can be simple values or in some cases references to distributed objects.

A related technology called *code on demand* transfers functionality between systems but neither computations nor threads. As the functionality transferred can subsequently be used for computations within the destination environments the technique can be helpful to implement

migration mechanisms. The best-known example of this technique is the Java programming environment [GJS1996].

Remote execution was pioneered by Stamos and Gifford in REV [StG1990b] and Falcone in HDS/NCL [Fal1987] both of which employ a lisp-like syntax for closures. Cardelli [Car1994] has based the object-based language Obliq on the transfer of closure which may also include references to remote objects and Fuchs has done so for the Scheme dialect Dreme [Fuc1995].

Some agent systems have been implemented using closures as well [Tsc1996b] and the POPCORN system [NL+1998] extends the notion to Java based computelets that are sent to remote machines for remote execution. An overview of the techniques related to the transfer of computations is provided by table 2.b.

	remote invocation	**remote execution**	**code on demand**	**thread migration**	**weak migration**	**strong migration**
Identity					(x)	x
State					x	x
Functionality		x	x		(x)	x
Computations	x	x		x		x

Table 2.b: Various techniques are related to the transfer of computations. Remote invocation and remote execution create new activation records within different environments. Thread migration complements weak migration through the transfer of computations of objects. The technique known as code on demand does not transfer computations but rather functionality that only subsequently can be used for computations within the destination environments.

Related to the concept of closures and strong migration are the invocation semantics that have been introduced in the context of Emerald [JH+1988] which migrate objects at the point of invocation through *call-by-move* or *call-by-visit* constructs either temporarily or permanently and create new activation records within the destination environment.

In the context of migration invocation semantics can be perceived as an continuum from call-by-value that uses copies of simple values to call-by-reference that employs references to objects including remote references, to call-by-visit where an object specified within an invocation is moved to location of execution, to call-by-move where an object remains at the new location of invocation even after the invoked operation returns [Jul1993].

Process Migration

The computations of processes are encapsulated by the address space abstraction except for active system calls. Asynchronous message passing or shared memory mechanisms that are used for communication between processes in most operating systems do also represent external computations of processes.

The address-space abstraction in combination with the process context of the operating system will be migrated during process migration. Whether the internals of the process are implemented in a single-threaded or multi-threaded way does not matter for a process migration mechanism as all internal computations are simply transferred as part of the address space.

The information about active system calls is held on a so-called system stack of activation records maintained by the operating system kernel. The transfer of the system stack involves the same difficulties as the transfer of non-encapsulated threads for system and language objects. Therefore, almost all process migration systems allow migration request to be handled only between system calls.

Only very few operating systems support the migration of processes while active system calls are being executed. As an extreme case, Accent [Zay1987a] also transfers the system stack as part of the process context to the destination and maps all dependencies of the process

accordingly within the operating system at the destination. Consequently, processes may be migrated in the middle of a system call and computations of the kernel can be continued at the destination.

System Migration

Large software systems are often multi-threaded and the system objects they use can themselves be single- or multi-threaded active objects. In addition to synchronous communication mechanisms, asynchronous message passing is used in some cases to transfer information between otherwise independent system objects. Similar to process migration system objects are usually only migrated if no active system calls exists for any of the internal threads of a system object.

Within multi-threaded environments as well as in the context of asynchronous message passing synchronization mechanism like monitors [Hoa1974] have to be applied before a migration can be performed. As an exclusive access to the object to be migrated is required by the thread that performs its migration all threads referencing the object have to be outside of critical sections.

If threads are transferred as part of system objects the references of activation records have to be able to cope with the implicit location changes. This can be done through implicitly memory mapping as in Cool2 [AJ+1992], through explicitly updates of references as in Commandos [CH+1993] or alternatively through proxy objects as in PEACE [Nol1994].

Software systems that do not use remote references internally do usually not support the transfer of computations. The first implementation of the Chorus Object-Oriented Layer (COOL) for example stops all threads of an system objects prior to migration and restarts them after migration has concluded at the destination [LeW1991].

Language Migration

Fine-grained language objects do in most cases not encapsulate computations themselves but take part in computations within their respective environments. Local and remote method invocation is the normal form of execution among language objects that are referenced as parameters or receivers of messages within activation records.

Only in the unusual case of asynchronous message passing between active objects like for example in Actors [AgJ1999] a language object encapsulates computations in the form of a single or multiple threads. The activation record of these encapsulated threads do nevertheless contain references to other objects that have been passed as parameters.

The transfer of computations during language migration requires the transfer of activation records of operations of the object to be migrated that have been invoked prior to the migration. Otherwise, the migrated object will have to be transferred back to the source as soon as the flow of control returns to these activation records.

References to the objects to be migrated of activation records that remain within the source environment need to be updated with the new location information. References to objects of the source environment within the activation records that need to be transferred have to be updated accordingly as well, like for example in Emerald [Jul1993].

As a result of the transfer of the activation records the flow of control will float between the source and destination environments when the previously invoked operations return as if these operation had been invoked via remote method invocation in the first place (see also chapter 4 page 179). Figure 2.f illustrates the transfer of activation records and the necessary update of references.

In multithreaded language environments migration operations have to be synchronized. This ensures that only a single migration is attempted for an object and that no other threads can access the object during migration. In the case of active objects the threads encapsulated by the object to be migrated will have to be transferred as a whole and the references of their activation records will have to be updated as well.

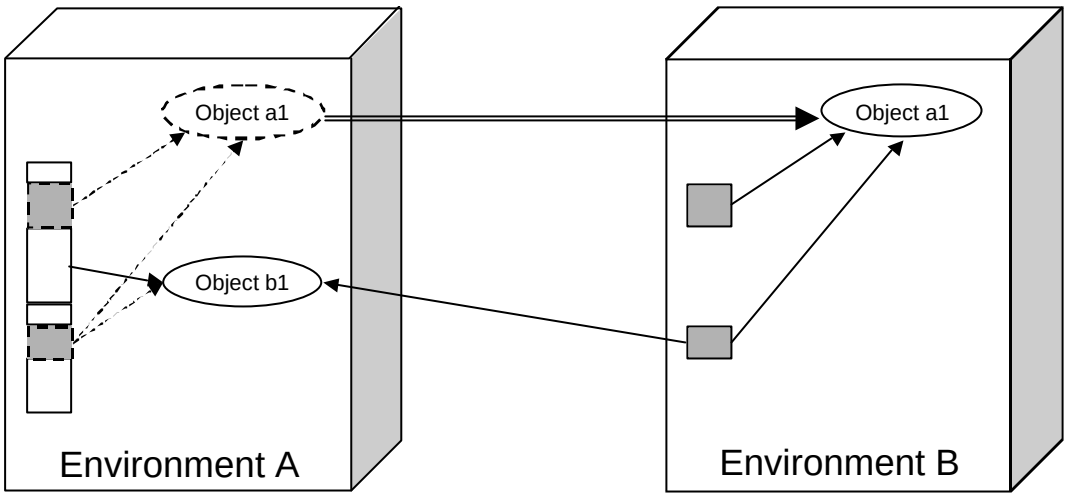

Figure 2.f: If computations of an object (shown here as ellipses) are migrated the activation records of invocations of the object (shown gray rectangles) are transferred to the destination environment. In the above example object a1 and its corresponding activation records are transferred. The references from activation records to objects are updated accordingly (the references between activation records are not shown to avoid confusion).

Due to the complexity of implementation only few language environments transfer computations as for example Trellis/DOWL [Ach1993a] while the majority of systems transfer only inactive objects. Surprisingly enough this statement does also hold for agent systems as most Java based systems transfer only inactive objects like for example ObjectSpace Voyager [Gla1998] and only newly implemented environments also transfer computations like Flage [TK+1995], Sumatra [ARS1996] or Telescript [Doe1996].

2.2 Migration Policy and Initiation

Migration systems, especially process migration systems, distinguish in most cases between a migration policy and a migration mechanism [ArF1989]. The *migration policy* determines when a migration has to be performed, which object should be migrated and whereto. The *migration mechanism* performs the actual migration of the corresponding unit of migration. Migration policies are also be implemented by some cases of system and language migration.

Migration policies are defined in accordance to the motivation why the migration of an object is desirable in the first place. The most common reason across all kind of migration systems is the maximization of resource utilization. Such an optimization can occur for example in the form of load balancing through process migration between machines. The *collocation* of objects achieves locality of access and reduces communication overhead. Another reason may be the migration of objects away from machines with imminent availability restrictions because of maintenance schedules for example.

Migration policies have to decide when to migrate what objects and to what destination. The actual migration mechanism has to decide what information has to be transferred and how this can be achieved. The progress of the migration is also checked by the migration mechanism as well as its completion since at least two separate environments have to be coordinated. The outcome of the migration, i.e. whether an atomic migration has completed successfully or not, can then influence subsequent decisions of a migration policy.

In contrast to the initiation through migration policies, migration can also be initiated explicitly by an application or by the object to be migrated itself. The reasons for the explicit initiation correspond to the rules that are used by migration policies [Sch1990] although they are probably more application specific. For example applications may be able to reconfigured themselves through object migration if their pattern of usage changes.

Regardless whether migration is initiated by a policy or explicitly a so-called *migration request* has to answer the following questions on behalf of the migration mechanism that will have to perform the migration:

- When ?

  A migration policy will initiate a migration request if the conditions implemented by the policy hold. However, a migration request may not necessarily be executed immediately by the migration mechanism. Depending on the implementation and due to various reasons a migration request may be rejected altogether or in some cases postponed until for example the execution of system calls has ended or the migration of dependent objects has concluded. The time of migration can therefore be interpreted only relatively in the sense of "as soon as possible".

- What ?

  The question "what to migrate" is determined in terms of the unit of migration. For example a load balancing policy has to choose a process to migrate in order to reduce the load on a particular machine. Even if the object to be migrated is specified explicitly an actual migration request may include additional information. The migration mechanism has to determine the set of objects that is actually migrated in response to a migration request as well as the amount of information that has to be transferred for each object.

- Where to ?

  The destination of a migration also has to be specified by a migration request. In the case of load balancing an analysis of load statistics is required to find a machine with a matching load pattern [Roe1998]. The destination can be specified explicitly within the migration request or implicitly as in the extreme case of the V-System that can use queries to determine destination nodes [TLC1985]. However, the destination environment has to be accessible by appropriate communication means at the time of migration.

Because a migration consumes a significant amount of system resources, migration policies also have to consider whether a migration is worth the effort. As a general rule, the time of continuous use of the migrated objects within the destination environment has to be longer then the time necessary to perform the migration. This rule is only a very rough measure as both time-factors can only be estimated at best.

The optimization of load distribution among a network of machines in general requires complete knowledge about the utilization of all machines involved. Ideally, the load any running process may generate in the future can also be estimated. Load balancing policies try to achieve as good approximations as possible using various techniques. The following quote [Nol1994 page 18][7] illustrates how important migration decisions are:

> “In the absence of hints about the usage of objects, the similarity of object mobility to demand paging of system with virtual memory [...] becomes obvious and the problems with regard to trashing are also comparable. Other approaches therefore try to formulate application-specific conditions that if applicable trigger a collocation of objects at a certain location [...]. This again bears similarities to the forward looking determination of a working set [...] in a virtual memory system .”[8]

Almost all process migration systems implement migration policies. The placement of processes through process migration is usually the real goal of the respective research effort. On the other hand, only few implementations of system migration use migration policies because migration decisions are made application specific. The same holds for most language migration systems.

---

[7] A similar quote can be found in [LeW1991].

[8] Translation of the following german text: "Hat man keine weiteren Hinweise auf die Benutzung eines Objektes, so ist die Analogie der Objektbeweglichkeit zum Demand Paging in Systemen mit virtuellen Speicher [...] allerdings offenkundig und die Probleme in Bezug auf Flattereffekte (Trashing) sind ebenfalls vergleichbar. Deshalb versuchen andere Ansätze, applikationsspezifische Bedingungen zu formulieren, bei deren Eintritt eine Zusammenführung von Objekten an einen bestimmten Ort stattfinden soll [...]. Letzteres hat wiederum eine Analogie zur vorausschauenden Bestimmung eines Working Sets [...] in einem System mit virtuellem Speicher.”

Process Migration

The main application of process migration systems is the global allocation of resources within distributed systems, especially the balancing of load between machines. The migration policy is usually enforced by the operating system itself or a specialized software entity thereof. Rhodos [SG+1996] for example uses a MigrationManager process to supervise process migration.

Process migration is only seldom initiated by a process itself (an exception is [Schu1990]) or by one user process for another. Process migration is usually initiated by a component of the operating system preemptively. Yet, a migration request may not be executed immediately as the process may be busy with system calls or input/output operations that have to be finished before migration can take place in most systems.

A migration request is therefore handled in the same way as other operating system events that need to be synchronized with the ongoing operation of the process. Once all operations with higher priority have been finished, the migration request is executed. Although not mentioned in the literature the usual mechanisms for deadlock prevention can be applied to migration requests as well.

Examples of user initiated process migrations are provided by the operating systems Sprite [DoO1990] and System V [TLC1985] that migrate processes to idle machines. Processes are migrated to different idle machines if the user of a particular workstations returns and begins to use the machine again.

System Migration

The migration of system objects is more guided by the logic of applications then by formal criteria. If available at all a migration policy for the migration of system objects is usually implemented as part of a supporting software layer within the overall software system. The migration policy is a function of the software system and as a consequence, system migration is sometimes even called application specific migration.

The collocation of data and computations is probably the main concern of the migration of system objects. For example, system objects that perform complicated operations on large amounts of data stored in databases or file systems benefit from migration to the machine that stores the data rather than from accessing the data remotely over distributed file systems. As system objects often cooperate intensively to fulfill a certain task, system migration is more frequently used for forward-looking reconfigurations of distributed systems rather than for measurement-based optimizations of load distribution [Sch1990].

Besides self-initiated and preemptive migration, that may occur at any time, system migration is sometimes implemented as cooperative migration that may only occur under certain conditions. Apart from the usual avoidance of system calls, the dependencies between system objects may create additional restrictions for the applicability of migration requests. In extreme cases, migration requests are queued by system objects and executed on a first-come-first-served basis, depending on the restrictions implied by the application logic.

Language Migration

Fine-grained migration is far less guided by migration policies then by explicit programmatic initiation. The *call-by-move* and *call-by-visit* constructs have been developed in the context of Emerald [JL+1988] to initiate object migration in the context of invocations. Nevertheless, in order to maintain locality of computations, collocations of objects also serve as the main reason for object migration at the language level among existing approaches.

A call-by-move permanently migrates the objects specified by the parameters of the call to the location of the execution of the invoked operation. In contrast, a call-by-visit will migrate the parameter objects back to their previous locations upon return of the result of the invocation. Apart from these invocation dependent migration constructs simple `migrate` or `go` statements are used to initiate object migration programmatically as for example in Telescript [TaV1996].

Agent system that operate at the language level in general implement different policy mechanisms as mobile agents are usually autonomous and decide for themselves where to

migrate next. Agents in Sumatra [ARS1996] can for example react to resource changes and effectively implement their own migration policy.

A completely different approach to the initiation of migration is implemented in the agent system Concordia [WP+1997]. An itinerary, i.e. a list of destinations to visit and actions to be performed is specified for each agent. An agent moves between the specified nodes and tries to perform the actions on the list for each corresponding node.

2.3 Migration Mechanism

The term migration mechanism denotes the actual implementation of the migration process that transfers objects between different environments. Depending on the level at which migration occurs and the unit of migration involved, the actual mechanisms are implemented quite differently. However all migration mechanism share some fundamental *prerequisites* that must be fulfilled before migration can be initiated and performed.

Beyond these basic prerequisites, several common forms of support by underlying software layers are used to implement the migration mechanisms. The term *support* is defined here as a feature of an underlying software layer without which a migration mechanism could still be implemented, albeit at higher effort. In contrast, migration would be impossible if the basic prerequisites are not fulfilled.

All migration mechanisms use some sort of abstraction for the actual transport of the unit of migration to its destination. Each migration mechanism employs an algorithm that implements the actual migration process depending on the available support from the underlying software layers. These building blocks of existing migration mechanisms are described in detail in the following subchapters.

2.3.1 Prerequisites

The prerequisites shared by all migration mechanisms are quite obvious. The source and destination environments must be able to communicate with each other. The destination environment must provide a receiving software entity that is able to participate in the migration process. This entity must be capable of recreating the transferred object in the context of the destination environment in such a way that it can be useful.

- Communication

  Some form of communication between the source and the destination environment is necessary in order to perform migration at all. While an inherent part of distributed systems and usually available in the form of network transports, this communication may also be performed on the basis of higher level protocols like distributed file access, inter process communication or remote method invocation mechanisms.

  Communication requires the destination environment to be addressable at least on the basis of raw network addresses, for example Internet addresses and port numbers of the popular TCP/IP protocol. Alternative addressing mechanisms which employ higher levels of abstractions as for example symbolic names can be used additionally.

- Receiver

  Often regarded as self-evident but nevertheless a crucial prerequisite to migration, a receiving software entity must exist within the destination environment that is able to perform all necessary steps of the migration process. The receiving entity has to be considered part of the migration mechanism. It has to take part in the migration communication, has to be able to interpret all representation formats used for the transfer of objects and has to be able to perform the appropriate actions during the migration process.

  Although only a receiving part of the migration functionality is required for a destination environment, most migration systems implement full sending and receiving migration capabilities within both the source and destination environments. Only few systems that use migration of processes for load balancing in an asymmetric way implement only receiving capabilities within destination environments as for example Locus [Thi1990].

- Dynamic Loading, Binding and Object Creation

  During the migration process the objects to be migrated will have to be recreated at the destination environment at runtime. If transferred as part of the migration process the corresponding object definitions have to be made available within the destination environment at runtime as well. A technique known as *dynamic loading* that enables new functionality to be added to an environment at runtime and is necessary in this regard.

  Dynamic binding allows the use of the transferred functionality by the migrated and newly recreated objects and is therefore required for any migration mechanism. The term dynamic binding is also used to describe the process of method invocation in the context of message passing in the sense that the decision which code is to be executed is postponed until runtime.

Apart from these basic prerequisites all other features of existing migration systems can be regarded as support of the particular software environment that can also be implemented by the migration mechanisms themselves. For example, full location transparency or automatic rerouting of messages eases the task of implementing object migration (see page 42).

Without the prerequisite available within the participating envrionments migration will not be possible. Without supportive features migration will still be possible albeit a lot more work will be required on behalf of an application developer in order to use a particular migration mechanism in the absence of such features.

Process Migration

Within process migration systems, communication between the participating nodes is performed via inter process communication or via raw network protocols. In some cases services of the operating systems, i.e. demand paging mechanisms are redirected between the source and destination machines as in Accent [Zay1987a].

In all known cases the process migration mechanism is part of the operating system and therefore readily available at the destination as a receiving entity. The management of migration is often performed by a dedicated component of the operating system as for example by the MigrationManager in Rhodos [PaG1996].

The rebinding of migrated processes is done through reestablishing the process context within the operating system at the destination node. This operation also makes the process itself available to an inter process communication mechanisms at its new location. Depending on the particular migration mechanism and the support by the operating system access to open files is also reestablished.

System Migration

Software systems establish communication via high level remote message passing mechanisms and to a lesser extent via raw network protocols. In most cases the addressing of system objects is already provided by the software systems themselves. In some cases even support for the management of location changes of system objects is available.

The migration mechanism is in most cases part of a supporting software layer assumed to be available at the destination especially within environments that use distributed shared memory techniques. Some software systems define a special software entity as the receiver for object migrations.

Dynamic binding is provided as part of the software system or by the language environments supported by the software system. Some environments that implement system migration are based on virtual machine abstractions and enable dynamic loading of new functionality while systems that work with traditional file based compilation usually require executables to be deployed prior to migration.

Language Migration

Systems supporting language migration also communicate either via remote method invocation mechanism or via raw network protocols. If not already covered by remote method invocation

mechanism that offer symbolic names, addresses of destination environments are implemented as native network addresses, especially for raw network protocols.

The migration mechanisms are either part of the language environments or built on top of these for example as supporting libraries. Special objects are sometimes defined as receiving and or sending entities for migration requests. In some cases, the migration mechanism is implemented at the application level as a library and becomes an integral part of an application.

Dynamic binding is in many cases already provided by the language environments or has been added to the language used as for example in Choices [CI+1993] that extends C++ with dynamic loading and binding that is used in the FreezeFree migration mechanism [Rou1995]. As not many language environments support dynamic loading, deployment of executables prior to migration required for language migration in many cases.

2.3.2 Support

Several common features of underlying operating systems, software services or language environments can be used as support for the implementation of migration mechanisms. Various kinds of so-called "transparencies" as well as several other forms of support can be identified. This discussion focuses on customary features and does not include some extreme cases like transactional communication mechanism [Lis1988a].

Transparencies

In the context of distribution the term *transparency* denotes the characteristic of a software system to be able to hide differences between local and remote operations. Transparency provides the illusion, that all operations are performed locally and hides the details of the management of remote operations from the developer and the user.

A definition of transparency in the context of migration has been paraphrased by Horn and Cahill [HoC1991 page 359]:

> "By transparent access we mean that access to an object via an object reference when that object is remote from the reference is syntactically equivalent to when it is local, and does not require code to be rewritten, recompiled or even relinked if the proximity changes."

Compete transparency of distribution is not desirable in the context of migration as the awareness of distribution is an inherent requirement of any migration mechanism. The availability of transparencies can nevertheless simplify several aspects of the implementation of migration significantly.

Several forms of transparencies [Bor1992] exist and some of them can be useful for the implementation of object migration. The most common forms of transparencies are location, access, invocation, concurrency, failure and replication. Additionally transparency of migration can also be defined.

- Transparency of Location

  Transparency of location is related to the addressing and naming of software entities within a distributed system as an object cannot determine whether another object it references is local or remote. A software environment that provides location transparency manages the necessary mapping between local references and remote locations of objects automatically. Although complex to be implemented transparency of location alone is not very useful without other forms of transparencies.

- Transparency of Access

  Transparency of access builds upon transparency of location as its provides location independent access to the state of objects. In addition to the location independent reference, access operations can be initiated and are performed in the same way regardless whether the object accessed resides locally or on a remote system. Information stored in a remote object is thus available in the same way as locally stored information.

- Transparency of Invocation

  Transparency of invocation enables that invocations for remote objects appear to be the same as for local objects. As invocations pass parameters and yield results, location independent references to the respective objects are used wherever the transfer of atomic values is not sufficient. Transparency of invocation is therefore dependent on transparency of location.

In the presence of transparencies the only noticeable difference for the developer or user of distributed applications is the additional time it takes to perform a remote operation in comparison to a local one. The other forms of transparencies, namely concurrency, failure and replication are of less interest in the context of migration.

With *transparency of concurrency* the result of a remote operation is the same whether the operation is carried out in parallel or not. *Transparency of failure* yields fault tolerance as the result of a remote operation is the same whether an error occurred during the communication for the invocation or not.

With *transparency of replication* an access to an object in a distributed system is performed regardless whether several physical copies of so called *replicas* of one logical object exist or not [SDP1991]. Replication can be seen as orthogonal to migration in the sense that replication implies a copy-operation while migration implies a move-operation. Replication requires that all replicas of an object are kept consistent and implies a significant network utilization through a synchronization protocol and. If replication and migration are combined replicas can be established through migration and can themselves be migrated.

Interestingly enough, migration can also be made transparent. The term *transparency of migration* denotes, that operations on objects can be invoked in the same way regardless whether the referenced objects have been migrated or not. In some cases, migration can even be made transparent to the objects that are migrated.

Transparency of migration can for example be implemented using proxy objects. An object that is migrated can be replaced by a proxy in the context of the source environment. The proxy object provides the illusion that the original object was not migrated at all. Figure 2.e illustrates the use of proxy objects for transparency of migration.

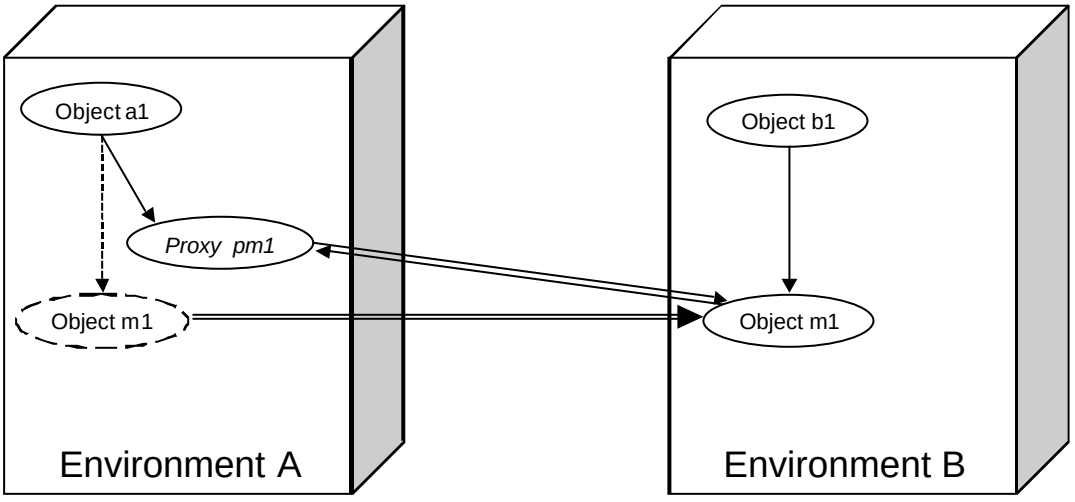

Figure 2.g: Transparency of migration can be achieved in the context of the source environment using a proxy object in place of the object being migrated (show here as ellipses). In the above example, proxy pm1 is substituted for object m1 that is migrated to environment B. After migration the reference from object a1 to object m1 is referring to proxy pm1, messages sent to proxy pm1 will be forwarded to object m1 in environment B and results will be returned vice versa.

Transparencies can be implemented differently depending on the characteristics of the particular operating system, software system or language environment and the communication

mechanism used. Either explicitly or implicitly remote references as well as proxy objects are employed and used by existing migration mechanism (see also chapter 4 page 158).

In all cases, the information about the distributed location of an object has to be updated if the object is migrated. The techniques that can be used to perform the required update are described in the following sections and can be called immediate update, forward pointers, home node, name server and broadcast.

- Immediate Update

  An *immediate update* can be performed when a migration request is processed through a direct update of all remote references of an object with the new location information. Although feasible only in environments where all remote references that point to an object are know an immediate update may still be prohibitively expensive to perform. If a remote reference is unreachable due to a network partition an immediate update operation will fail at least partially.

- Forward Pointers

  A *forward pointer* refers from the source of a migration to the destination and can be followed by an object that tries to access the migrated object at its old location. Sophisticated mechanisms have been developed to update remote references in the context of chains of forward pointers. Although an update can be postponed in some cases until possible network partitions have been overcome, forward pointers are still susceptible to failures if the chain of forward pointer is broken due to malfunctioning nodes.

- Home Node

  A *home node* can be used for an object as a central point of access. With each migration the new location information of an object is stored at its home node and other objects can update their remote references via the home node. The home node is in most cases the location where the object has been created, but reassignment of home nodes is also supported by some systems. Unfortunately, the home node is a single point of failure in the case of a malfunction.

- Name Server

  A *name server* can be used to store location information for each registered object and can resolve queries for objects when remote references become outdated. The use of a name server requires the assignment of unique names or identifiers to objects in order to distinguish them from each other. Name servers can also be clustered into a federated group of mutually updated servers in order to avoid a single point of failure.

- Broadcast

  A *broadcast* sends a query for an object to all known hosts of a system, one of which will answer with the correct location information. As a broadcast usually requires an expensive network operation it is most often used only as a last resort. If no host answers a request in a certain time frame the particular object has to be regarded as unavailable. In the case of network failures the necessary retries will increase the already costly overhead of broadcast operations.

Object migration mechanisms can take advantage of some forms of transparency. Especially beneficial to object migration are transparency of location, access and invocation, as objects are moved between environments but related objects can remain at their locations. However, full transparency is not desirable in the context of migration as the location of each object has to be determined if necessary.

Transparency of failure can be helpful in order to prevent inconsistencies if network failures occur while object migrations are in progress. Transparency of concurrency and replication are less beneficial as migration mechanism must be able to access objects exclusively and unambiguously. Table 2.c provides an overview of the usage of transparencies among migration systems.

Transparency	Process Migration	System Migration	Language Migration
Location	x	x	()
Access			()
Invocation	x	x	(x)
Failure	()	()	()

Table 2.c: The different forms of transparencies are available to migration mechanisms to a varying extent. Location transparency is available to most migration systems with only sparse support by language environments. Transparency of access at the object level is usually not available for process and system object and only supported by few language environments. Transparency of invocation is generally available in the form of inter process communication and remote method invocation mechanisms respectively. Transparency of failure is only seldom implemented directly but optionally available in the context of transaction services.

The use of transparencies by migration mechanisms is limited and usually guided by the availability of the respective services in the particular operating systems, software systems or language environments. Only few migration systems implement transparencies as part of the migration mechanisms themselves.

Cross-platform transparency mechanisms as for example CORBA [Sie1996] that provides transparency of invocation can be used for system and language migration on the basis of supporting libraries in combination with simple development tools like preprocessors. The CORBA lifecycle service [PeG1999] in particular supports location changes and uses a name-server approach to update location information.

Process Migration

Process migration systems usually make use of transparencies provided by the underlying operating system as in System V [TLC1995]. Transparency of location is available among distributed operating systems in terms of globally unique process identifiers as well as transparency of invocation in the form of location independent inter process communication mechanism [PG+1996].

Location and access transparency is implemented in Charlotte [ArF1989] through named inter process communication links rather then host-addresses. Some process migration mechanisms as Accent [Zay1987a] provide limited transparency of failure as they are able to cope with the malfunction of a destination host during migration.

Sprite [DoO1990] achieves location, access and migration transparency through redirection via a home node. A Sprite process may migrate any number of times but can still be addressed via its home node i.e. the node its was originally started on. The migration is also transparent to the migrated process which is not able to determine that it is not running on its original host.

Different kinds of update techniques are used by process migration systems. Immediate update of location information is performed by Charlotte [ArF1989] where both ends of communication links are known. However the designer of Charlotte regret this decision as Artsy and Finkel [ArF1989 page 53] have stated:

> "In retrospect we would have used a different [transparency] implementation for Charlotte [...]. We would have used hints for link addresses, which are inaccurate but can be readily checked and inexpensively maintained, rather than using absolutes, whose complete accuracy is achieved at a high maintenance cost"

Immediate update is also used by Locus [Thi1991] where a pipe with multiple readers and writers is coordinated via a single storage site that is informed of migrations of participants. Forward pointers are used for example by DEMOS/MP [PoM1983] and broadcasts are used in

System V [TLC1985] to publish the new location of a migrated process. The FreezeFree migration mechanism implemented on top of Choices [RoC1996] also sends notifications of location changes.

System Migration

Software systems that provide migration for large grained system objects usually employ their own location transparency mechanism based on global name spaces. These are mapped to actual network addresses through specialized name servers. Transparency of access and invocation are also often provided as a native implementation but transparency of failure is not commonly implemented. Transparency of migration is for example implemented in Electra [Maf1993a] via smart proxies that hide migrations of system objects.

The main location update technique used among system migration environments are forward pointers as implemented in Commandos [HoC1991] in conjunction with name servers for finding previously unknown objects. The concept of a home node is also used by agents systems as in the Mobile Object Layer [CB+1998].

Language Migration

Language environments are usually not designed with distribution in mind and instead highly optimized for local operations. As a consequence, transparencies are only seldom "built into" these environments, with the notable exception of Emerald [Raj1991]. However, in several cases transparencies have been successfully added on top of existing languages as in Beta [BrM1993] or Smalltalk [GaY1993].

Location update techniques used by language migration systems include forwarding pointers as used by Emerald [JL+1988] and Trellis/DOWL [Ach1993b] as well as in the agent system like Sumatra [ARS1996]. The home node concept is used in Mozart [RB+1998] and in DC++ [ScM193b] where the home node of an object may be changed if the object is moved between DCE domains.

Emerald [JL+1988] implements broadcasts of queries as a last resort if forward pointers are unavailable due to node crashes. Emerald also introduced unavailability handlers that work like exception handling mechanisms in the case that the location information can still not be found even after a broadcast operation.

Other forms of support

In addition to the transparencies described above several other features of underlying operating systems, software systems or language environments are used by migration systems. Apart from the extreme case of hardware support within shared memory multiprocessors all other services are software based. However, not all available means of support like distributed shared memory, distributed file systems, inter process communication, remote procedure call facilities, and persistency mechanisms are used by all migration systems.

- shared memory multiprocessors

Shared memory multiprocessors (SMM) implement common address-spaces for high performance parallel computing through specific hardware architectures for tightly coupled processing units. These architectures allow address-spaces to be swapped between individual processors of a multiprocessor system.

It appears questionable whether such an extreme utilization of specialized hardware should be considered as process migration. Support for shared memory multiprocessors can rather regarded as a mere machine specific scheduling mechanism as Smith [Smi1988] states:

> „Process migration is most interesting in systems where the involved processors do not share main memory, as otherwise the state transfer is trivial. A typical environment where process migration is interesting is autonomous computers connected by a network“

Although included in this overview as one extreme form of support for process migration, shared memory multiprocessors do not affect issues of heterogeneity of hard- and software since only

for very specialized operating systems are used for migration among homogeneous hardware. As a consequence shared memory multiprocessors will not be further discussed.

- Distributed Shared Memory

The technique of *distributed shared memory* (DSM) has been developed for loosely coupled networks of processing units like workstation clusters. Distributed shared memory enables memory segments to be mapped consistently across distributed address-spaces as in Amber [CA+1989b]. The technique is bound in most cases to the size of memory pages and to the use of homogeneous hardware and operating systems.

Distributed shared memory can be used by migration mechanisms to transfer all necessary kinds of information during migration through shared memory regions. The restriction of the memory transfer to the size of memory pages limits the usefulness of this technique for finer grained objects as well as the performance of the information transfer.

- Distributed File Systems

Distributed file systems (DFS) provide location independent access to secondary storage devices. In most cases distributed file systems provide transparency of access to files stored on remote machines or central file servers. Implementations of DFS are achieved through Client/Server protocols as well as through various forms of caching schemes combined in some cases with replication or even voting algorithms [Bor1992].

Migration mechanisms often use distributed file systems as common repositories for shared functionality or as a means for the transfer state and control information. Some process migration mechanisms use distributed file systems to store and transfer checkpoint images of processes during migration, in some cases in cooperation with the demand paging mechanisms of operating systems.

- Inter Process Communication

Almost all operating systems implement some form of *inter process communication* (IPC) mechanisms that allow information to be transferred between processes. IPC is established through signals, i.e. software interrupts, pipes, i.e. buffered streams of bytes, shared memory regions or other techniques as well as combinations of the aforementioned.

For most distributed operating systems inter process communication mechanisms have been extended to work also between remote machines. In a few cases all communications between user processes and the operating system are performed through IPC, including input/output operations and file handling.

Inter process communication can be used by migration mechanism for the transfer of information during migration. Some examples of process and system migration use IPC mechanisms of the underlying operating systems. Language migration systems do not use inter process communication mechanisms due to the relative high communication overhead for fine grained objects implied by the necessary system calls.

In the context of process migration IPC itself is affected by migration. Since migrating a process takes a significant amount of time the operating system can in most cases not deliver IPC messages that are sent by other processes. Either the operating system has to handle this situation or other processes need to be informed that the process being migrated is unavailable.

Depending on the operating system implementation, IPC messages for a process in transit have to be either queued at the sender of the message, the source of the migration or the destination of the migration. When the migrated process becomes available again the queued messages have to be forwarded. Alternatively the delivery of IPC messages may also be rejected and has to be resent by the originating process.

After migration, the target of an IPC communication may reside at a different location and the communication requests have to be redirected. As mentioned before this can be done through forward pointers from the source of a migration, symbolic names that can be resolved via name server or broadcasts of queries can be used to update the location information either by the inter process communication mechanisms themselves or by the sender of an IPC message.

- Remote Procedure Call and Remote Method Invocation

Many language environments can be extended in the context of distribution with the technique of *remote procedure calls* (RPC). In the case of object-oriented language environments an analog technology has been developed in the form of *remote method invocation* (RMI) that extends local method invocation in the context of distribution.

Systems that offer transparency of invocation implement either technique but implementations of RPC or RMI are not necessarily transparent. Some systems distinguish between local and remote invocations and message that are sent to remote systems have to be constructed explicitly (see also chapter 4 page 158 for a detailed discussion).

Although the availability of non-transparent remote method invocation is better than no support at all, migration mechanism will become more complicated if remote method invocations have to be managed explicitly. Even worse, the object to be migrated will have to be designed with both local and remote operations in mind, which will prohibit the use of migration in some cases.

- Persistency

Some object-based environments use *persistence* in order to extend the lifetime of objects beyond the execution time of the programs they were created with. Objects are made *persistent* by saving their state to secondary storage media for later retrieval. The persistent object storage mechanism can either be implemented by the objects themselves or by the corresponding environments. Alternatively, object database management systems (ODBMS) can be used that also provides additional services like indexing of the stored data.

Persistency can be used for the transfer of the state of objects during migration of system or language objects [SiL1996, SiS1997]. An object that is made persistent during migration is stored in a serialized format that can be read by the destination environment in order to recreate of the migrated object. Some object database management systems support even heterogeneous environments through a platform independent storage format and perform the necessary conversions as part of the save and load operations. Figure 2.f illustrates the use of persistency in the context of migration systems.

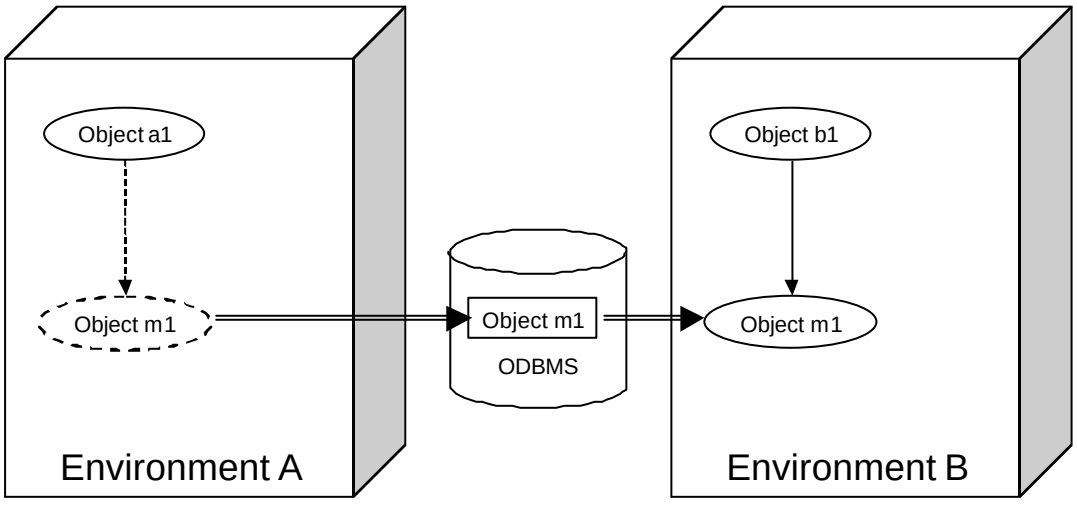

Figure 2.h: Persistent object storage mechanisms can be used to migrate the state of objects between environments. In the above example, object m1 is transferred to environment B using a serialized representation (shown here as a rectangle) that is stored by a persistent storage mechanism. The object is retrieved from its persistent storage upon recreation of the migrated object within the destination environment.

Only the state of objects can be transferred via a persistency mechanism. The behavior of objects is usually assumed to be available from within the application program that makes use of the persistent object storage mechanisms. The application code therefore has to be deployed at the destination environment prior to migration before the persistent storage can be used for migration purposes.

- Network Transport

In general, migration mechanisms can be built on top of network transports without any further support by the operating systems, software systems or language environments. Under these circumstances, a migration mechanism has to implement its own communication protocol as well as its own representation of the objects for the transfer over the network.

Despite the additional effort many language based migration mechanisms and even some system and process migration mechanisms, are implemented using raw network transports. The main reasons are lack of appropriate support as well as the need for specifically optimized protocols.

The use of support technologies by migration mechanisms not only depends on the general availability of the supporting software layers but also on significant advantages for the migration process. Table 2.d provides an overview of the use of software support by migration systems.

	Process Migration	**System Migration**	**Language Migration**
Shared Memory Multiprocessors	()		
Distributed Shared Memory	(x)	()	()
Distributed File Systems	(x)	(x)	(x)
Inter Process Communication	(x)	()	
Remote Procedure Calls		(x)	(x)
Remote Method Invocation		(x)	(x)
Persistence		()	()
Network Transport	()	(x)	(x)

Table 2.d: Migration mechanisms make different use of available support technologies. Shared memory multiprocessors are the only example of hardware support, available solely for process migration. Distributed shared memory is often employed in process migration and to some lesser extent in system and language migration. Distributed file systems are utilized by most migration systems. Inter process communication mechanisms are mainly used for process migration, remote procedure call mechanisms for system migration and remote method invocation mechanism for language migration. Persistence is occasionally found in system or language migration. If no other support is available native network protocols are used.

Most migration systems utilize only the support readily accessible to them from existing software layers. If adequate support is not available simple alternatives are implemented in most cases. Only few migration systems implement generally applicable solutions for missing support functionality. However, some innovative approaches implement support functionality from the outset especially if they were intended to break new ground.

Process Migration

Process migration mechanisms are usually based on the support provided by the underlying operating systems. In most cases, inter process communication facilities are used to initiate and

control the migration process and for the transfer of process state and context as in Accent [Zay1987a]. System V [Che1988] even redirects all input/output operations via IPC.

Some process migration mechanisms use distributed file systems for the transfer of snapshots of address spaces sometimes in conjunction with demand paging mechanisms as for example in Sprite [DoO1990]. In some cases, distributed shared memory mechanisms are also used for the transfer of process state.

During migration, pending inter process communication requests can be queued and forwarded by the source of a migration as in Amoeba [SZM1994], DEMOS/MP [PoM1983], Rhodos [PG+1996] and System V [TLC1985]. Alternatively, the sender of an IPC requests can resent their messages as in Charlotte [ArF1989].

System Migration

Some large software systems implement distributed shared memory mechanisms as support for system migration. Remote procedure call mechanism and occasionally inter process communication are used to control the migration process. Distributed file systems and persistency mechanisms are employed to transfer the functionality and sometimes also the state of system objects between environments.

The software system Guide [BB+1991] based on Commandos [CH+1993] uses persistence to migrate objects and a combination of persistence and memory mapping is used in the agent system PLANET [K+1999b] to transfer agents between environments. Agent systems also use persistence as for example the Open Services Model (OSM) [LG+1997] that creates profiles, which include state and functionality of agents.

The handling of remote procedure calls and remote method invocation during migration are often implemented by the particular software systems themselves. COOL [HMA1990] for example implements transparent message forwarding in its communication layer. The agent system Concordia [WP+1997] also queues messages for agents.

Language Migration

Language environments that implement migration, apply remote method invocation mechanisms to control the migration process and to transfer fine-grained objects. In some cases persistence mechanisms are used to help in the transfer of object state and distributed file systems are used to make the functionality of objects available. Distributed shared memory is only rarely applied because in most other cases basic network transport services are used.

For example the parallel processing language Orca [BKT1992] uses distributed shared memory to migrate passive objects between threads that run on different processors. Language independent runtime support for distributed shared memory has also been proposed with the Tarmac [LuA1990] system that could be employed by migration mechanisms.

2.3.3 Abstractions

Migration mechanisms need to transfer the state, functionality and computations of objects between the participating environments. A variety of abstractions is available and different abstractions can be used for each kind of information although not all formats are suited equally well in all cases.

The spectrum of available formats reaches from hardware oriented binary representations, through virtual representations for virtual machine implementations and intermediate representations used in language processing, to actual source code as well as at least theoretically to formal abstractions.

- binary

  A *binary representation*[9] provides no abstraction or the most primitive one, that is stored in the main memory of a particular machine and can be interpreted by a particular processing

[9] The term “binary” may be misleading as all information is binary coded within today's computer systems. A more precise definition would be “machine interpretable representations of state and functionality”.

unit directly. Binary representations depend on the hardware architecture they where designed for and can be used directly for migrations among homogeneous environments. State, functionality and computations can all be represented in binary form.

While being specific for a certain kind of processing hardware the main advantage of binary representations is that the performance of the direct execution by the *central processing units* (CPU) is preserved. Changes to binary representations, for example the update of references, are on the other side more tedious than in other representation formats.

- virtual

  A *virtual representation* is a more abstract yet still machine-oriented form of representation used in the context of virtual machines as implemented by systems like Smalltalk [GoR1983] or Java [GJS1996]. A *virtual machine* is an efficient and portable native machine code program that interprets so called *virtual instructions* or *byte codes* that are generated instead of native machine code from a source program. The interpretation of these virtual instructions is usually at least an order of magnitude slower than the direct execution of native code by a particular processing hardware.

  The biggest advantage of virtual machine architectures is the platform independence that is favored by many migration mechanisms. While only functionality is represented by virtual abstractions, state and computations of the respective objects are dependent on the definition of the virtual machine as well.

- intermediate

  An *intermediate representation* also achieves platform independence. Used at various stages in the process of program compilation in the form of *abstract syntax trees* (AST) it represents the state of compilation before a decision about code generation for a particular hardware platform or virtual machine has been made.

  Intermediate representations can be interpreted directly or compiled into native code. Some hardware independent distribution formats are based on similar representations that are compiled into native machine code during load time. Intermediate representations are used mainly to represent functionality and in some cases computations in the form of closure.

- source

  A *source representation*, i.e. the textual source code written in a human legible programming language, can also be used as an abstraction. Source representations are used mainly for the transfer of functionality. State and computations can also be represented textually but require conversions to and from their in-memory representations in binary form.

  The source code transferred between environments has to be compiled at the destination into a native representation. Source code representations are therefore directly dependent on a particular programming language and limited by the availability of the corresponding tools at the destination.

- formal

  A formal representation can use specification techniques like state machines, petri nets, higher order logic, denotational semantics or other mathematical formalisms to provide an abstraction for migration as well. The biggest advantage of such formal abstractions is that certain characteristics of these representations can be mathematically proven.

  Unfortunately, their use for practical purposes is limited as formal representations are in most cases only available for special research oriented environments. Their usage is therefore limited to dedicated software environments or specific problems of common interest that can be formalized platform independently.

The existing migration mechanisms employ the whole spectrum of available representations. However not all representations are used at all levels and only few system use several formats

for different purposes. Table 2.e provides an overview of the use of abstractions by migration systems.

	Process Migration	System Migration	Language Migration
Binary	x	(x)	(x)
Virtual		(x)	(x)
Intermediate			()
Source	()		()
Formal			()

Table 2.e: Various abstractions are used by migration mechanisms for different purposes. Binary representations are almost universally used for process migration systems and to a lesser extent for system and language migration. Virtual representations are frequently used for system and language migration. Only few examples exist for the use of intermediate representations in language migration. Source code representations are sometimes used for language migration and in an extreme case also for process migration. Finally, formal representations are used in extreme cases of language migration.

The use of abstractions by migration mechanisms is mainly guided by the level and unit of migration. Binary representations are the default representation for process migration and different kinds of representations are used for various purposes in the context of system migration and language migration.

Process Migration

Process migration systems are usually based on binary representations since processes are defined by machine dependent in-memory images of address-spaces. Other abstractions would at least theoretically also be feasible for process migration as well but no actual examples could be found. One extreme form of process migration at least proposes a generated source representation for the migration of a process that includes its computational state [ThH1991].

System Migration

Software systems that provide migration also use binary representations for large grained objects, especially if memory-mapping techniques are employed. Some examples of system migration however use virtual abstractions both internally and for the transfer between nodes as for example Commandos [CH+1991] that specifies a generic runtime environment. The agent system Omniware [LSW1995] also uses virtual code to migrate system objects.

Language Migration

Language based migration mechanisms take advantage of all forms of abstractions. Binary representations are employed as an effective way for the transfer of state as for example in Emerald [Jul1993]. Virtual abstractions are used by language environments that are based on virtual machines like Distributed Smalltalk [GaY1993]. Agent systems exist that use virtual byte code as well, like for example Concordia [WP+1997].

Only few language migration systems use intermediate representations for the transfer of functionality and computations as for example Obliq [Car1994]. Source code representations are mainly employed for the transfer of functionality as in REV [StG1990a]. Some extreme cases apply formal abstractions even for migration of functionality as in the agent system TNET [HCS1997] where state machines are used to capture the semantics of agents. Another example is the π-calculus that is used in some systems like Seal [ViC1998].

2.3.4 Algorithms

The very heart of every migration mechanism is the migration algorithm that performs and controls the migration process. The migration algorithm is invoked after a migration has been

initiated through a migration request. Almost all migration algorithms used by existing migration systems implement in some way or another the following three obvious steps:

1. a representation of the object to be migrated is generated at the source environment
2. the representation is transferred to the destination environment
3. the object is made available at the destination and its remains are deleted at the source.

The typical migration algorithm starts after the initiation of migration through a migration request that specifies at least the object to be migrated and the destination it should be migrated to. As a first step, a representation of the unit of migration is generated at the source that encompasses the information necessary to recreate the object at the destination. One or several of the available abstractions can be used for the representation.

The generated representation is transferred to the destination through an available communication medium. In some cases, the transport mechanism is able to cope with communication failures and to ensure a reliable transfer of the representation. In any case, a migration algorithm must be able to determine whether a transfer could be completed or not. The migration algorithm has to either retry the migration or abort the migration and make the object available again at the source.

If the communication succeeded, the transferred representation is used to recreate the corresponding object at the destination, possibly using functionality already available at the destination or retrieved from other sources like distributed file services. The migrated object is then made available at the destination using appropriate support like transparencies, and continues its operation at the destination. The completion of migration is reported to the source environment, which will then delete what is left of the original object.

As part of the migration algorithm both source and destination environments need to cooperate. Some steps of the migration algorithms are performed at the source while others have to be performed at the destination. In most cases, the source will control the overall progress of the migration process and will give up this control only at the very last moment, when the object becomes available again at the destination.

Process Migration

Various forms of optimizations have been developed for process migration systems in order to minimize the latency of migration, i.e. the time a process is unavailable due to its migration. Most optimized process migration algorithms employ memory transfer strategies in cooperation with the "demand paging" memory management of the underlying operating system.

A direct transfer of the address-space is used for process migration in Amoeba [SZM1994], Charlotte [ArF1989] and Locus [Thi1991]. A combination of the transfer of the process state to a file server with a subsequent fetch of necessary pages by the destination is used in Sprite [DoO1990]. An algorithm called pre-copying is used in System V [TLC1985] that incrementally copies dirty pages to the destination until a threshold is reached and the transfer is finally completed. Lazy-copying is used in Accent [Zay1987a] where only few pages of the process are transferred at first and the remaining ones are copied as needed through demand paging.

The goal of the optimizations of the migration algorithms is to decrease the latency of migration, i.e. the time a process is unavailable. While the variations of the demand paging mechanisms used work in parallel with the continued execution of the process at the destination they also create unwanted residual dependencies (see also page 54).

The process migration algorithm with the shortest latency is the FreezeFree algorithm [RoC1996] implemented for Choices [CI+1993] that only transfers one code, stack and heap page during migration and does not freeze inter process communication at all. However comparatively long residual dependency are created as dirty files are flushed to a file server and pages will be demanded from the source until they flush operation has ended.

System Migration

Large software systems that provide migration for large grained objects often employ a memory-mapping phase before or after the transfer of a usually binary representation. The functionality of the migrated object is either assumed to be deployed prior to migration or needs to be made available at the destination before the object will be useful.

The same holds for objects that are transferred using virtual representations. For example, Commandos [CH+1991] transfers a binary representation of the state of system objects via a shared memory mechanism and relies on the availability of the virtual representations of the corresponding functionality through a distributed file system.

Language Migration

Algorithms for the migration of language objects often have to consider dependencies between objects before a representation can be generated and transferred. The functionality of the object being migrated is either assumed to be available at the destination or will be transferred as well. The recreation of the object at the destination involves the dynamic binding of the object and its functionality within the respective language environment.

Exceptions to the usual migration algorithm can be found as well. A migration mechanism that uses the CORBA lifecycle service shares the state of objects during migration, which means that the object is available during migration at both source and destination environments and kept consistent through a synchronization protocol [Peg1999].

Some agent systems do also employ different migration algorithms as for example PLANET [KM+2000] that uses asynchronous message passing between agent environments. An agent is transferred via memory mapping to a persistent store and on to the destination but may be already destroyed at the source while the destination may not even have started to receive it.

2.4 Properties of Migration

Migration mechanisms can be characterized through several properties of implementational, conceptual and theoretical nature. Beyond characteristics of the initiation and completion of migration, issues of consistency and fault tolerance are considered as well as the general applicability of the particular migration mechanisms.

- preemptive

  The initiation of migration can be characterized as *preemptive*, when a migration can take place at any time. Most migration systems however defer migration requests until for example system calls or input/output operations have concluded. Some systems allow migration requests only at so called *migration points* in the program code [Ple1996].

  Migration points are sequences of machine instructions that do not conflict with the execution of a migration request, for example the code between system calls. Other migration systems restrict migration to inactive objects, i.e. only objects which do not take part in any computations at the time a migration request is received can be migrated.

- atomic

  The migration of an object is usually expected to be *atomic*, i.e. the object is transferred to the destination either completely or not at all. This property is also known as "at most once" semantics. Some migration mechanisms are able to detect and handle communication failures and network partitions as well as node failures themselves.

  Unfortunately, many migration mechanism compromise this intended transactional nature of migration due to performance considerations or limitations of the implementation. The importance of atomicity with regard to process migration is emphasized by Artsy and Finkel in the design of Charlotte [ArF1989]:

  > “Migration may fail in case of machine and communication failures, but it should do so completely. That is, the effect should be as if the process were never migrated at all or, at worst, as if the process had terminated due to machine failure”

- residual dependent

  Most optimization techniques used for migration create so-called *residual dependencies*, i.e. dependencies of the migrated object on the availability of services of the source environment. In the worst case, the migrated object will not be functional within the destination environment if the source environment becomes unavailable. Residual dependencies are unwanted side-effects as stated by Steketee et al. [SZM1994 page 197]:

  > "In general residual dependencies are undesirable because of the chain of dependencies when a process is migrated several times and the continuing use of resources on the source machine. This has detrimental effects on both performance and reliability."

- fault tolerant

  The property of fault tolerance offered by some migration mechanisms, is related to atomicity. In contrast to the "all or nothing" approach of atomicity, fault tolerant mechanisms are able to recover from failures during a migration and continue the overall process eventually leading to a successful migration.

  The disadvantage of fault tolerance is the performance degradation as well as the required additional resources resulting from for example additional retries of network communication or for example from the redundant storage, for example within a logging mechanism, of the information that is transferred during migration.

- symmetric, transitive

  One theoretical characteristic of a migration mechanism is the nature of sequences of migrations it is able to perform. A migration mechanism is called *symmetric* if an object that has been migrated from a source environment to a destination environment can also be migrated in the opposite direction.

  A migration mechanism is called *transitive* if objects can be migrated from environment A to environment C, when migrations from environment A to environment B and from environment B to environment C are possible. Although almost trivial for many existing migration systems these theoretical properties are much more difficult to fulfill in the heterogeneous case.

Apart from the obvious conflict between fault tolerance and performance the ideal migration mechanism is at least atomic, symmetric and transitive. Characteristics like preemptivity, fault tolerance as well as the lack of residual dependencies are only rarely achieved. Table 2.f provides an overview of characteristics of migration systems.

	Process Migration	**System Migration**	**Language Migration**
preemptive	(x)	()	()
atomic	(x)	(x)	(x)
fault tolerant	()	()	()
residual dependent	(x)	(x)	(x)
Symmetric	(x)	(x)	(x)
Transitive	(x)	(x)	(x)

Table 2.f: The properties of migration mechanisms differ. Preemptive migration is seldom found among migration systems other then process migration. Most migration systems aim for atomicity but only few are able to work fault tolerantly. Residual dependencies are common among migration systems. Almost all migration systems operate symmetrically and transitively with only few exceptions.

The properties of migration mechanisms vary considerably among existing systems with different consequences for the objects being migrated. Whether a particular property can be achieved may also depend on the individual objects to be migrated as well as on the participating environments, especially in the heterogeneous case.

Process Migration

Preemptive migration is the norm for process migration systems with the exception of active system calls. Higher prioritized system calls will be finished before migration requests are serviced. Most process migration mechanisms implement atomicity because they abort the migration if the destination host fails. However only few systems provide fault tolerance if, for example a source node fails.

A major issue of process migration are residual dependencies usually with regard to the process context within the source environment, if some resources still need to be available. Continuous access to open files can be achieved through distributed file systems and network connections as well as inter process communications can be rerouted, but direct hardware access can never be migrated. Some optimization techniques also create residual dependencies at lest temporarily.

Surprisingly enough not all process migration systems operate symmetrically or transitively although most systems do. The residual dependencies created by many process migration mechanisms prevent or at least impede the symmetric or transitive migration of process objects after an initial migration. If residual dependencies exist only temporarily subsequent symmetric or transitive migrations become possible after a latency period.

For example Amoeba [SZM1994] avoids residual dependencies and passes a token to ensure that at most one copy of a migrated process is running. However, the loss of a process in migration cannot be avoided if the source host fails during migration. Charlotte [ArF1989] avoids residual dependencies as well.

Temporary residual dependencies are created in MOS [BaL1985] as any part of the process can remain at the source and is fetched only on demand. A temporary residual dependency on the virtual memory image of the source process is also created in Accent/Spice [Zay1987a] as well as for the access to files of the source environment which are accessed via the IPC mechanism. The same holds for System V [TLC1985] that performs file input/output operations via inter process communication as no distributed file system is available.

Sprite [DoO1990] explicitly employs residual dependency on the home node for every migrated process in order to achieve complete migration transparency. No other residual dependencies exist if a process is migrated again to another node that is different from the home node. In Sprite some part of a migrated process always remains at the source and the source acts as a paging device for the migrated process.

System Migration

Software systems providing migration for large grained objects usually operate non-preemptively because the interactions of the various system objects can be fairly complex, especially in the multi-threaded case. Atomicity of migration is in most cases achieved through an abort of the migration in the event of communication failures.

System migration mechanisms also have to address issues of fault tolerance and residual dependencies as system objects are often large and dependent on one another. Symmetric and transitive migrations are rare in reality because system migration is mostly used for sets of objects to be collocated rather than for frequent moves of objects. Residual dependencies are created by some system like Amber [Ca+1989b] that depends on the home node for the lookup the new location of a migrated object.

Language Migration

In many cases, migration of fine-grained language objects is initiated during normal operations of objects, hence non-preemptively. Preemptive migration is only rarely available especially in

the context of the transfer of computations. Although language migration is usually implemented atomically, fault tolerant implementations are an exception.

Language migration does usually not lead to residual dependencies apart from the normal level of remote references in distributed environments. Symmetric and transitive migrations are normal properties of language migration mechanisms. Fault tolerance of object migration is handled to some extend in Emerald [Jul1993] through the use of availability handlers.

2.5 Heterogeneous Object Migration

Most migration systems assume that migration only takes place between homogeneous environments. *Homogeneous migration* is assumed to be possible by definition and no checks whether migration can be achieved or not will be performed prior to the execution of a migration request or as part of the migration mechanism.

Migration systems that support *heterogeneous migration* among different environments consider in most cases only hardware differences. These differences are addressed mainly through a virtual machine implementation as in Commandos [CH+1993] and only in few cases through the use of sophisticated conversion techniques before or after the transfer of object representations as in Emerald OS [StJ1995].

In general, several levels of heterogeneity between environments can be determined with different consequences for migration. To address the different levels of heterogeneity, various strategies can be identified some of which are in use by existing systems. Other approaches to overcome heterogeneity can be envisioned at least theoretically.

2.5.1 Levels of Heterogeneity

The characterization of two environments as homogeneous or heterogeneous depends on the level of abstraction as well as on the degree of detail used. While homogeneity of systems can be defined as absolute equivalence in all aspects, this definition would be useless in practice as all systems would then be heterogeneous. Even otherwise identical systems differ at least in the state of their respective computations as well as some basic configuration parameters like network addresses.

Heterogeneity in the context of migration has to be considered either from the standpoint of the object to be migrated with regard to what is required by that object or as the more general question of compatibility between environments. While the first rather narrow viewpoint will be sufficient for the migration of a single object, a broader analysis will be necessary in order to determine the general migrateability of sets of similar objects between environments.

At least four different levels of heterogeneity between environments can be identified. These are the hardware platform, the operating system, the programming languages and the libraries used to implement particular applications. The application level has to be added for completeness. As the objects to be migrated are part of the respective environments they are also affected by these levels of heterogeneity.

- Hardware

  The most obvious differences of hardware platforms are determined by the central processing units (CPU) used, that require their own data and instructions set formats. A particular binary representation can be directly interpreted by the central processing units either directly though logic circuits or indirectly through a microprogram.

  The spectrum of hardware differences ranges from simple byte ordering through the representation of data, e.g. the binary format of floating point numbers to the overall architectures of the processing units that require different machine instruction sets as well as particular code optimization techniques.

  Other less obvious differences of hardware platforms are memory management schemes and the handling of input/output devices as well as other various forms of connections of peripherals. These are usually hidden from software systems through abstractions like operating systems calls or primitives of language environments.

- Operating System

  Operating systems differ in their architectures, the concepts they use and the services they provide. Their functionality is made available to the developer of applications through system calls or in the case of object oriented operating systems (OOOS) through methods of corresponding system objects.

  The diversity of the programming interfaces and the different conventions of parameters as for example incompatible security concepts are a major source of problems for migration mechanisms between heterogeneous operating systems. Concepts like access rights or directory structures can usually not be matched without some loss of fidelity.

- Language

  Programming languages show a great deal of heterogeneity. Programming paradigms including declarative, logical, functional, procedural and object oriented programming differ in the general methodology of problem solving. Even languages of the same paradigm differ significantly. Object-based languages, for example, implement built-in functionality, base types, control structures, exception mechanisms and concepts like inheritance, or delegation in distinct ways.

  Different implementations of the same programming language may be heterogeneous as well. The memory layout and the structure of activation records as well as the code optimizations applied may differ significantly enough, that a direct binary transfer is impossible despite the fact that the same source program is used.

- Library

  Libraries provide fundamental functionality for the construction of software systems. Several layers of libraries are often used to structure large systems into several increasingly finer levels of abstractions. In the context of object-based languages class libraries are applied for the construction of complex object definitions from simpler ones.

  The runtime environment of a particular programming language is usually augmented by a set of libraries that provide functions like memory management or input/output primitives. Basic operating system services like system calls are made available through libraries as well as higher-level services like graphical user interfaces. Application level libraries provide implementations of common data-structures like linked lists as well as access to additional software services like database management systems.

  The use of different sets of libraries may create heterogeneous environments even when all other levels of abstractions including the programming language implementation are homogeneous. On the other hand, the use of top-level libraries that abstract from implementation details and provide common functionality can make heterogeneous libraries appear to be homogeneous. The use of language independent libraries requires common calling conventions and memory management techniques that are not always shared in the context of the same operating system or processing hardware.

- Application

  Heterogeneity of applications has to be considered if objects of one application have to be migrated into another application. Apart from other forms of heterogeneity applications employ different abstractions like data-structures for the particular solution they provide. In order to migrate objects between different applications, a minimum of conventions as for example general access methods to fundamental objects are necessary for the migrated objects to be useful within the destination environment.

Some forms of heterogeneity like heterogeneous load, i.e. differences of resource utilization among otherwise homogeneous environments, cited by some author of migration systems [Ste1998m, Röd1998] will not be considered here. While being an essential factor for decisions of migration policies especially for load balancing, heterogeneous load has no relevance for the more principle question whether an object can be migrated between two different system at all.

Although all forms of migration can be confronted with all levels of heterogeneity not all combinations of heterogeneity and migration are addressed by existing migration systems that support heterogeneity. Table 2.g provides an overview of the support of different kinds of heterogeneity by migration systems that were designed with heterogeneity in mind.

	Process Migration	**System Migration**	**Language Migration**
Hardware	(x)	(x)	(x)
Operating System		(x)	(x)
Programming Language		()	
Libraries		()	()
Applications		()	()

Table 2.g: Not all levels of heterogeneity are addressed by existing migration mechanisms for heterogeneous environments. Heterogeneous hardware is supported by some process, system and language migration systems. Heterogeneous operating systems are addressed by system and language migration on the basis of libraries that provide standardized APIs. Heterogeneous programming languages are supported by few system migration mechanisms. Heterogeneous libraries and applications are supported by even less examples of system and language migration.

Most migration systems that support heterogeneity address only one level of heterogeneity that is typically not identical to the level of migration. Only very few systems address more than one level of heterogeneity. Of those migration mechanisms that do address heterogeneity, differences of hardware and operating systems are the main targets.

Process Migration

Process migration systems are only affected by heterogeneous hardware platforms and operating systems. Heterogeneity of programming languages, libraries and applications are hidden by the address-space abstraction of processes. Of the few process migration systems that address heterogeneity all concentrate on the problem of process migration between heterogeneous hardware platforms as Theimer and Hayes [ThH1991], Arabe et al. [AB+1995], Pleier [Ple1995] or Shub [Shu1990]. None attempts to perform migration of processes between heterogeneous operating systems.

System Migration

Large software systems providing migration of system objects are affected to a different extent by all levels of heterogeneity. In order to avoid heterogeneity, migration mechanisms for system objects usually depend on a common set of libraries across all supported languages and operating systems. Some system migration mechanism are based on a single operating system but support a predefined set of language environments.

System objects that can be migrated are designed with the respective common programming interfaces in mind. If heterogeneity of hardware is considered it is addressed through the use of virtual machine abstractions as in Commandos [CH+1993]. No system migration mechanism could be found that is able to migrate between different software systems. Migration between heterogeneous agent systems has been considered at the level of system objects [GSC2000].

Language Migration

Language environments usually hide details of operating systems from the developer through the definition of programming primitives and standard libraries. If the objects to be migrated are not dependent on specific operating systems features, heterogeneity at the operating system

level does usually not appear as an issue. Differences of libraries and applications are addressed through additional higher-level libraries that provide common functionality.

Language migration is nevertheless affected by heterogeneous hardware as different machine code is generated for each platform. Virtual machine abstractions are the most prominent means to overcome heterogeneous hardware at the language level, especially among agent systems some of which are based on the Java virtual machine [GJS1996].

Migration of fine-grained objects between heterogeneous language environments is considered only sparsely as for example by the agent systems Ara [PeS1997] and Planet [MMK1998]. However, none of these systems works without significant changes to the existing language environments.

Heterogeneous libraries are also seldom addressed. For example DC++ [Scm1993a] restricts the use of different class hierarchies to the homogeneous case and Bennett [Ben1990] considers different approaches to address heterogeneous libraries for Distributed Smalltalk but implements only a homogeneous scheme.

2.5.2 Approaches to Heterogeneous Migration

The term *heterogeneous migration* can be applied to migration mechanisms that addresses aspects of heterogeneity. Various techniques to overcome heterogeneity have been developed for existing migration mechanisms with different motivations in mind and under different preconditions and circumstances.

In order to abstract from the details of the implementation of each relevant migration mechanism the following characterization attempts to categorize the existing approaches in a consistent framework. This also opens up the possibility to identify opportunities for new approaches to overcome heterogeneity.

Due to the fact that the destination environment differs from the source environment and the object to be migrated is initially a part of the source environment, an approach to overcome heterogeneity can either be applied to the object being migrated in order to adjust it to the conditions of the destination environment or the destination environment has to be modified in order to be able to accommodate the object to be migrated.

A particular approach to overcome heterogeneity can either be performed prior to migration, for example through the implementation of the prerequisites of a migration mechanism within a particular environment or through changes made at the time of migration either to the object being migrated or at least theoretically to the destination environment.

The question when a particular approach can be applied also influences the kind of objects that can be addressed by a particular heterogeneous migration mechanism. A priori changes can only be made for whole categories of objects with common characteristics. In contrast, migration time changes may be done in reaction to an actual migration request depending on the objects to be migrated or on the combination of source and destination environments.

The approaches available prior to migration can be characterized in order of increasing complexity as *restricting* the object to be migrated and to some lesser extent of the participating environments, as *adjusting* the object to be migrated to the conditions of the destination environment, as *embedding* common functionality within both environments together with a common implementation of the object to be migrated, and as *enabling* the destination environment to provide the necessary functionality.

The migration time approaches can be characterized in order of increasing complexity as *conversion* of the representation of the object being migrated, *extension* of the destination environment for example with additional functionality, *adaptation* of the object being migrated to the condition of the destination, and as *transformation* of the destination environment.

Despite the fact that any combination of these techniques can be used by a migration mechanism, each approach will be discussed independently. Table 2.h offers an overview of the applicability of techniques to overcome heterogeneity with regard to the software entity the technique is applied to and ordered by the phase of migration the technique can be performed.

		Source	Object	Destination
a priori	**Restricting**	(x)	x	(x)
	Adjusting		x	
	Embedding	(x)	(x)	x
	Enabling			x
migration time	**Conversion**		x	
	Extension			x
	Adaptation		x	
	Transformation			()

Table 2.h: The applicability of the various approaches to heterogeneous migration is shown in relation to the software entities involved. Prior to migration the "restricting" approach can be applied to the object being migrated and to some lesser extent to the destination environment. Adjusting the object to be migrated prepares it for the destination environment. Embedding and enabling change the environments during the implementation of migration and in some cases the object to be migrated. Conversion and adaptation change the object during migration while extension and transformation change the destination environment at the time of migration.

Each approach to overcome heterogeneity has its own characteristics and can be applied under different circumstances. None of these approaches is able to address all levels of heterogeneity or all different aspects of heterogeneous systems. Each of these approaches has its individual advantages as well as drawbacks.

The following sections provide a description and an assessment of each possible approach to overcome heterogeneity that can be applied prior to migration:

- Restricting

  The simplest technique that can be used to overcome heterogeneity prior to migration is the *restricting* approach that addresses the functionality of the object to be migrated. A common denominator of functionality among the participating environments is determined and the resulting restrictions are applied to the object to be migrated and to some lesser extent to the destination environment.

  Although applicable only at the software levels and certainly not to overcome heterogeneous hardware, this technique can be very effective for a wide range of applications. Unfortunately, the restricting approach does not scale very well with the number of different heterogeneous environments involved as the common denominator tends to become quite small. The approach is also limited in the sense that the participating environments have to be known a priori and additional environments cannot be added subsequently except for already compatible cases.

- Adjusting

  The *adjusting* approach changes the design or the implementation of an object prior to migration in such a way that it is able to work within the destination environment. These adjustments may change the functionality of the object significantly in contrast to the restricting approach that only limits the design to the use of common functionality.

  Although very powerful as heterogeneous operating systems, languages, libraries and application can be addressed, adjusting the object to be migrated requires knowledge about the destination environments prior to migration. Changes that have to be made to the object may also complicate the design and implementation of the object to be migrated. Adjustments required by different destination environments may also be conflicting which can limit the applicability of the approach significantly.

- Embedding

  The most common technique used to address heterogeneity at the hardware level is the *embedding* of a common functionality within both the source environment and the destination environment to be used by the migrated objects. Various forms of virtual machines are widely used examples of this technique.

  Embedding can overcome heterogeneity of hardware and operating systems and to some extent even of languages. Embedding essentially creates a homogeneous environment for the object being migrated within otherwise heterogeneous destination environments. It requires significant effort from the developer as the embedded functionality has to be implemented for each participating environment, which limits the scalability of the approach.

  Furthermore, the object to be migrated has to be designed and implemented for the embedded environment or has to be ported to the embedded environment prior to migration. In order to create the object to be migrated at the source, the embedded environment has to be available within the source environment too.

- Enabling

  An unusual approach to overcome heterogeneity is the *enabling* of the destination environment with functionality required by the object to be migrated. In contrast to the embedding approach, enabling does not merely embed a closed environment within the destination environment but changes the environment accordingly. For example, an object-based programming language that does not support dynamic binding at runtime will have to be enabled to do so in order to support object migration.

  Enabling can be applied at the application and library as well as to some extent at the language and operating system level. The enabling approach is limited in the sense that the functionality to be implemented by the destination environment has to be known prior to migration.

  The effort necessary to apply enabling depends on the complexity of the changes that have to be made. The enabling approach is limited in the number of source environments it is able to support by the possible conflicts that can arise from the different functionality to be added in order to accommodate different kinds of objects.

The following approaches to overcome heterogeneity can be applied to the objects to be migrated during the migration process:

- Conversion

  The most common approach to heterogeneity applied at migration time is the *conversion* of the representation of the object being migrated either prior or after the transport between environments. Conversion implies only representational changes but not changes of the functionality of objects.

  Often performed in the presence of heterogeneous hardware, conversions can become prohibitively complex if other levels of heterogeneity are involved. If conversion is applied at the binary level, full information about the construction of the binary representations, for example memory layouts or the format of activation records have to be known in order to perform the necessary conversion steps. The number of different environments that can be supported is limited by the effort necessary for each environment.

- Extension

  The *extension* approach can be used at the level of heterogeneous libraries or applications when functionality required by the object being migrated is not available within the destination environment. If a suitable common representation exists and the destination environment is capable to make the required functionality available then the required functionality can be added as an extension to the destination environment.

  The extension approach is limited by possible conflicts with existing functionality at the destination as well as by potential conflicts arising from consecutive migrations.

Heterogeneity of hardware can only be addressed for the functionality of objects and its applicability to heterogeneity of operating systems is also limited to special cases.

- Adaptation

  The *adaptation* of the object being migrated to the conditions of the destination environment extends the conversion approach. In contrast to the conversion approach, which only applies changes to the representation being transferred, adaptation will change the functionality of the object being migrated. If for example a datastructure required by an object is not available, the object being migrated can be adapted to use a similar structure available at destination that provides equivalent or even extended services.

  This approach can be used for migrations between environments with heterogeneity at the application, library and to some extent even at language operating system level. Although quite powerful conceptually, adaptation will be limited to well known cases in practical implementations. Its use to address heterogeneous operating systems will also be limited due to the complexity.

- Transformation

  A merely theoretical approach to heterogeneous migration at the software levels is the *transformation* of the destination environment at runtime in order to provide missing functionality for the object being migrated. Included in this overview only for completeness, the transformation approach appears to be unrealistic. Even if feasible, changing the destination environment at migration time may simply be unreasonable. Except for extreme cases where objects with a very long lifetime are to be migrated the delay necessary for the transformation will simply be prohibitive.

As mentioned previously, not all approaches to overcome heterogeneity can be used to address all kinds of heterogeneity. Table 2.i provides an overview of the possible applicability of the approaches to overcome heterogeneity for the different levels of heterogeneity.

	Hardware	**Operating System**	**Language**	**Library**	**Application**
Restricting		(x)	(x)	x	x
Adjusting		x	x	x	x
Embedding	x	x	x	x	x
Enabling		()	()	(x)	(x)
Conversion	x	(x)	(x)	(x)	(x)
Extension	(x)	(x)	(x)	x	x
Adaptation		(x)	(x)	(x)	(x)
Transformation	()	()	()	()	()

Table 2.i: The applicability of approaches to overcome heterogeneity for the different levels of heterogeneity. Restricting of objects to operating system or language functionality is only possible if a common denominator exists. Adjusting can be applied at all software-related levels and embedding also on the hardware level. Enabling has to be applied with care. Conversion is mainly applied to hardware differences and extension to heterogeneous libraries and applications. Adaptation will work only in a well-known context and transformation may not be applied at all.

Some of the presented approaches have obvious limitations, for example, changes to the destination environment are not possible in the context of heterogeneous hardware. Restricting will not work in the context of heterogeneous operating systems and languages if the source and destination environments do not share the necessary common functionality.

Conversion is mainly applied in the context of heterogeneous hardware alone as it becomes more and more complicated when used with additional levels of heterogeneity. All other approaches may quickly lead to conflicts if applied in combination. Table 2.j provides an overview of the actual and potential use of approaches to heterogeneity.

	Process Migration	System Migration	Language Migration
Restricting	(x)	(x)	x
Adjusting	x	(x)	x
Embedding	(x)	x	x
Enabling	(x)	(x)	(x)
Conversion	x	(x)	x
Extension	(x)	(x)	x
Adaptation	(x)	(x)	(x)
Transformation	()	()	()

Table 2.j: The use of approaches to overcome heterogeneity among migration systems differs. An “x” indicates the actual usage of an approach by a migration system, while its inclusion in parentheses “(x)” indicates only a potential and parentheses alone “()” a very unlikely usage.

Only few approaches to overcome heterogeneity are actually found in existing migration mechanisms that address heterogeneous systems. The most frequently used approaches are embedding and conversion. Other techniques to overcome heterogeneity are often hard to implement and therefore applied only seldom or not at all.

Process Migration

Several approaches to heterogeneity have been used by process migration systems in different contexts, namely restricting, adjusting and conversion. For example restricting programs to use only a common set of functionality is a well known portability technique that is applied for example in conjunction with like standards the POSIX interface to operating system services.

The adjusting of a process implementation to be able to be migrated between heterogeneous platforms has been exemplified by Dome [AB+1995] that uses source code instrumentation to generate cross-platform compatible checkpoints. Pleier [Ple1995] combines that with the restricting approach in order to define migration points that are compatible across environments and with the conversion approach for the necessary transfer of state during migration.

Although the use of embedding techniques would be feasible in the context of process migration through the use of an embedded operating system no example that uses this approach could be found. The enabling approach is frequently used among members of the Unix family of operating systems although only for portability and not for migration purposes.

Conversion is used for heterogeneous process migration in a modified version of System V [Shu1990] that initiates process migration non-preemptively and uses executables deployed prior to migration with multiple code formats and identical memory layout across all platforms. The address-space of a process is converted to the format of the destination hardware.

The extension of an operating systems functionality at runtime has been implemented by the Exokernel [EKO1994] although not for migration purposes. Adaptation of processes for heterogeneous migration has been proposed Theimer and Hayes [ThH1991] but was not implemented. No use of the transformation approach for process migration could be found.

System Migration

Software systems that perform migrations of large grained system objects use embedding in the form of virtual machines in order to overcome heterogeneity of hardware and operating systems as for example in Commandos [CH+1993]. Some agent systems like Strat0Sphere [WAA1998]

that implement agents as compound objects use the Java virtual machine [GJS1996] for the same purpose.

The agent system Ara [PeS1997] integrates interpreters for different languages through a common run-time system that has to be embedded into each participating execution environment. Restrictions are imposed for the individual agent code that are intended to guarantee compatibility of migration between the different execution environments.

Enabling and extensions can also be used on the basis of programming interfaces but will only be useful if the heterogeneous systems share common functionality. All other techniques are probably feasible in principle but no example of an existing system that uses these techniques in the context of system migration could be found.

Language Migration

Migration mechanisms for fine-grained language objects use a combination of approaches to address various aspects of heterogeneity. Restricting is applied [ScM1993a] as well as extension [Ben1990] in the context of heterogeneous libraries. Conversion is used to address heterogeneous hardware directly as in OS Emerald [StJ1995].

A combination of approaches is employed by the agent system Planet [KM+2000] that uses conversion between native and canonical representations of state, functionality and computations in order to migrate agents between execution environments. Planet also adjusts mobile objects through code instrumentation to use memory-mapping primitives and applies the restricting approach through a common API [KM+1999b] for operating system access and enforcement of security domains.

Through the use of virtual machine implementations embedding is applied by may migration systems in the context of heterogeneity of hardware, operating systems and to some extent languages. The Java virtual machine [GJS1996] is used by the majority of agent systems for that purpose. All other approaches though feasible in principle are not employed by any of the systems investigated.

2.5.3 State of Heterogeneous Object Migration

Existing migration systems that address heterogeneity focus mainly on one or two levels of heterogeneity, in most cases heterogeneity of hardware and operating systems. Only few systems address migration among heterogeneous languages, libraries and applications additionally or exclusively.

Popular approaches to overcome heterogeneity are embedding of homogenous environments prior to migration as well as conversion of object representations during migration. Also applied often are the approaches of restricting and adjusting prior to migration in combination with conversion during migration. Other approaches are only seldom used.

The most frequent model of heterogeneous migration that can be called the *embedding model* is based on an embedded virtual machine that overcomes heterogeneity of hardware and operating systems. Using this model an application that uses migration is deployed across the participating environments. The internal representation of the objects to be migrated within the embedded environment is transferred during migration.

Second frequent model that can be called the *adjusting model* uses the approach of restricting and adjusting prior to migration for the development of an application and the generation of appropriate code for each participation environment. During migration, a system specific representation is transferred and converted in accordance with the destination environment.

Table 2.k provides an overview of the levels of heterogeneity that are addressed by existing migration systems as well as the usage of approaches to overcome heterogeneity. Agent systems have been included in this analysis as they employ similar techniques although with different objectives.

Among systems that use the embedding model most are based on Distributed Smalltalk or Java. Notable exceptions are Obliq [Car1994] that uses its own engine to interpret abstract syntax trees as well as Telescript [Doe1996] that uses an interpretive language and TNET

[HCS1997] that is based on a state machine model. One implementation of Distributed Smalltalk [Ben1990] addresses also heterogeneity of libraries though the extension of destination environment with additional functionality.

	migration			information				heterogeneity					prior to mig.				during mig.			
	Process	System	Language	Identity	State	Functionality	Computations	Hardware	Operating System	Language	Library	Application	restricting	adjusting	embedding	enabling	Conversion	Extension	Adaptation	Transformation
(Pleier)	x			x	x	x	x	x					x	x			x			
(Theimer, Hayes)	x				x	x	x	x					x	x			x			
D.Smalltalk (Bennet)			x	x	x	x		x	x		x				x			x		
DC++			x	x	x						x		x	x						
Dome	x			x	x			x	x				x	x						
Oblig			x				x	x	x						x					
OS Emerald			x	x	x	x	x	x					x	x			x			
System V (Sub)	x			x	x	x	x	x					x	x			x			
Agent Systems																				
MOL			x	x	x	x		x	x						x					
Odyssey			x	x	x			x	x						x					
Omniware		x				x		x	x	x						x	x			
OSM			x	x	x	x		x	x						x					
Planet			x	x	x	x	x	x	x	x				x		x	x			
Sumatra			x	x	x	x	x	x	x						x					
Telescript			x	x	x	x	x	x	x						x					
TNET			x	x	x	x	x	x	x						x					

Table 2.k: An overview of existing heterogeneous migration systems and related mobile agent systems. The level migration and the kind of information transferred are displayed as well as the level of heterogeneity addressed and the approaches to overcome heterogeneity prior or during migration.

The adjusting model is used for process migration. Applications are designed specifically for migration in Dome [AB+1995] that inserts the necessary code in to the application source to generate checkpoint information at defined migration points. The code for the recreation of the execution state from the checkpoint information on other platforms is generated as well. A similar technique is used by Pleier [Ple1996].

A modification of System V by Shub [Shu1990] as well as a modified version of Emerald [StJ1995] use conversion to migrate processes and language objects respectively in the context of heterogeneous hardware. Both systems convert the binary representation of the state the functionality and the computations of the objects being migrated.

Only few examples of other approaches to overcome heterogeneity exist. Omniware [LSW1995] uses a machine independent binary representation to transfer functionality between heterogeneous hardware and operating systems. As the machine independent format is compiled into native machine code at load time, Omniware is also able to combine functionality from different programming languages, which have to be enabled prior to migration with the necessary functionality.

The agent system Planet [MMK1998] uses a canonical representation to capture the state, functionality and computations of objects that can be migrated in the context of heterogeneous hardware, operating systems and language environments. During the migration process the canonical representation is converted into the native format of the destination environment, which needs to be enabled prior to migration to perform the conversion.

Limitations and Drawbacks

All the existing migration systems that address heterogeneity as well as the approaches to overcome heterogeneity they employ exhibit different limitations and drawbacks. None of the system investigated is able to address all levels of heterogeneity and only few able to operate without changes to the participating environments.

The embedding model is able to cope with heterogeneous hardware and operating systems but is not well suited to address heterogeneous languages and offers not support for heterogeneous libraries and applications. The necessary implementation of the embedded environment limits the number of environments that can be supported. The confinement of the objects that can be migrated to functionality that is offered by the embedded environment is artificial and prohibits the use of functionality that is available in particular host environments.

The perceived omnipresence of the virtual machine based Java programming language [GJS1996] does not mark the end of innovation in migration technology as it limits any further migration technique to applications that are designed and implemented for the Java programming environment. Apart form the ongoing discussion about the obvious execution overhead, the virtual machine approach renders some potential of object migration to overcome other forms of heterogeneity meaningless.

The adjusting model on the other side requires significant changes to the source code of the objects to be migrated and as a consequence renders the use of widespread development tools like symbolic debuggers impossible. It is inherently limited in the number environments that can be supported due the possible dependencies of the generated code.

The approach of adjusting the objects to be migrated is also related to the conversion of the object representation during migration. The more adjustments are applied prior to migration the less conversions have to be made during migration and vice versa, but both approaches do not scale well in terms of heterogeneity.

The more levels of heterogeneity are addressed the more complex the combined use of the adjusting and conversion approaches gets. The adjusting model works well if a single level of heterogeneity like hardware is addressed. Combinations of different levels of heterogeneity like hardware and operating systems or even language environments are prohibitive to address using the adjusting model due to the complexity of the code generation and the necessary conversions.

Migration systems that use other approaches to overcome heterogeneity or combinations of different approaches also suffer from limitations. The restricting approach for example is not able to address heterogeneity of hardware, can only be applied prior to migration and does not scale with either the number of levels applied to or the number of environments involved in migration.

The enabling approach requires changes to existing environments in order to be able to host the objects to be migrated within the destination environments. Apart from being able to work only in combination with restricting, adjusting or conversion the enabling approach is also limited in the number source environments that can be supported and does not scale well with the number of levels of heterogeneity addressed.

The extension approach is able to address heterogeneity at the library and application level and only in combination with the conversion approach also heterogeneity at the hardware and operating system level. It does not require changes to either the object being migrated or the participating environments and scales well with the number of different environments supported.

The adaptation approach requires changes to the objects being migrated and is able to address all levels of heterogeneity with the exception of heterogeneous hardware that can only be addressed in combination with conversion. It is limited in the number of different environments that can be supported due to the individual adaptations necessary for each platform.

The transformation approach requires changes to the participating environments at runtime and is able to address all levels of heterogeneity at least theoretically. However, no existing migration system could be found that uses the transformation approach. And the approach does

probably not scale well, neither with the number of different environments nor with the levels of heterogeneity supported.

The last three approaches to overcome heterogeneity are only used by few existing migration systems and are therefore not well understood. Apart from the unrealistic transformation approach, they are at the same time due to their characteristics the most promising approaches for further research.

Research Opportunities

The analysis of the different existing approaches to heterogeneous migration reveals the following opportunities for further research:

- None of the existing migration systems is able to address all levels of heterogeneity not even with combinations of different approaches to overcome heterogeneity.
- None of the existing migration systems is able to work without changes either to the participating environments or the objects to be migrated.
- None of the existing migration systems is able to scale with both the levels of heterogeneity and the number of different environments supported.
- Heterogeneity at the level of languages, libraries and applications is addressed by only very few existing migration systems
- Approaches to overcome heterogeneity during migration are used by only few existing migration systems

The focus of the rest of this work lies on the development of a new migration mechanism for language objects that is able to address all levels of heterogeneity and that is able to work with existing environments. The mechanism should be open to all approaches to overcome heterogeneity. The mechanism should provide a framework for further experimentation and exploration of new approaches to heterogeneous migration.

3 Heterogeneous Object Migration at the Language Level

The design of a migration mechanism for language objects that works with existing environments and is able to addresses all levels of heterogeneity is influenced by a lot of options that can be chosen as well as tradeoffs that have to be made. The following chapter describes the design of a novel migration mechanism that is named *Heterogeneous Language Migration* and will be referred to in the following as the HLM migration mechanism.

This chapter outlines the decisions that shape the design of the HLM migration mechanisms as well as its prototypical implementation. The design follows a pragmatic approach towards overcoming heterogeneity at all levels. Some of the limitations of the HLM migration mechanism that result mainly from its focus on existing language environments may lead to the development of advanced migration techniques that are outlined in chapter four.

3.1 A Pragmatic Approach

Migration between heterogeneous language environments is confronted with a great variety of differences on several levels of abstraction. Disparate programming paradigms, differences of language concepts and constructs, of object definitions and representations as well as of details of the implementations of language environments need to be considered.

The more general question of migration between language environments of different programming paradigms, e.g. between declarative and procedural languages, lies clearly beyond the scope of this work. Whether migration across language paradigms is possible at all will remain an open research question although some promising indications exist [Weg1987]. This work is focused on language environments based on object technology.

Language environments within the realm of the object-based or object-oriented programming paradigm implement various concepts of object technology in different ways. Some may also add new variations or invent additional concepts. Even among languages that support the same concepts object definitions and representations differ significantly. A general migration mechanism would have to determine whether the concepts of the environments participating in a migration match, whether the implementations of these concepts are compatible, to what extent they differ and how migration can be performed.

Such a comparison will not be possible in the general case regardless whether it would be based on a mathematical formalism or on the actual implementations. For example two methods can not be tested for equivalence of their behavior due to the problem of computability. The same applies to more complex concepts of object technology like method lookup algorithms or asynchronous message passing mechanisms.

Rather then trying to find a theoretical all encompassing mechanism for migration of objects between all kinds of language environments, a pragmatic approach is followed here. Guided by a number of design objectives for the migration among heterogeneous languages in general, only a limited set of concepts and constructs will be supported by the migration mechanism to

be designed. The complexity of this migration mechanism is further reduced through a limitation of the concepts that can be supported and through prerequsites for the environments.

The pragmatic approach taken here, does not attempt to provide the theoretical basis for a migration mechanism of objects between all possible language environments. Instead the design of the mechanism is focused on the implementation of migration between existing heterogeneous language environment. The pragmatic approach is expressed in the objectives of the HLM migration mechanism, the limited the set of supported concepts and the prerequisites for the environments.

The HLM migration mechanism addresses specifically the transfer of object semantics among heterogeneous language environments with limited knowledge about the destination and without requiring changes to the participating environments. The HLM migration mechanism aims to achieve consistency of object semantics at the level of object definitions and consistency of object state at the level of object representations in the context of applications designed for the mechanism.

The HLM migration mechanism and its prototypical implementation is intended as a proof of concept and as a testbed for further research in heterogeneous migration. Various extensions to the HLM migration mechanism that lift some of the restrictions imposed for the mechanism can be made to widen the set of supported environments. Some of these are outlined in chapter four and may lead to the development of future versions of the mechanism. The objectives that guide the design of the initial version of the HLM migration mechanisms, the concepts that are initially supported and the prerequisites are described in the following subchapters.

3.1.1 Objectives

The objectives that guide the design of the HLM migration mechanism emphasize applicability and ease of implementation. The goal is to maximize the applicability of the HLM migration mechanism without requiring changes to existing language environments. As a tradeoff a number of limitations apply, for example, not all language concepts can be supported.

The following list of design objectives is ordered in terms of the importance of the individual objective starting with the most important one. Throughout the design process minor objectives have been overruled by more important objectives. Each entry in the list describes a particular objective and discusses its characteristics and dependencies.

1. No Changes to existing Environments

The HLM migration mechanism should be able to add migration to existing object based environments without implications for these environments. No fundamental changes of a language definition or the implementation of the corresponding language environments should be necessary in order to implement the mechanism. Especially no language syntax or compiler changes should be required.

Migration should be perceived as an extension of the existing language system that can be incorporated into any application. Migrations that would require changes to the environments will therefore not be possible through the HLM migration mechanism. The migration of computations for example is not supported by the HLM migration mechanism, as changes to the participating environments would be necessary (see also chapter 4 page 179 for a detailed discussion).

As a consequence, applications that use the HLM migration mechanism will have to be designed specifically for migration. Migration of objects of existing applications can not be supported by the HLM migration mechanism unless they are changed accordingly. Due to these consequences the approach to migration exemplified by the mechanism can be called *migration by design*.

2. Minimal Assumptions about Destination Environments

The HLM migration mechanism should be able to operate with only minimal assumptions and limited knowledge about the destination environments, especially with regard to the functionality available at the destination. All information about the destination that is

necessary to perform a migration should be gathered at the time of migration and should be used only for consecutive migrations. The HLM migration mechanism should especially not require the distribution of functionality or other kind of information among the participating environments prior to migration.

This objective stresses the universal applicability as well as the dynamic and spontaneous aspect of the HLM migration mechanism. For example the availability of object definitions within a particular destination environment may change over time due to development work within that environment. Information once gathered for destination environments may become invalid as these environments evolve over time. On the other hand optimization of the performance of the migration mechanism, although not a major goal, can be enhanced through the use of various caching schemes (see chapter 4 page 152).

3. Maximum of Functionality

The HLM migration mechanism should be able to convey as much functionality as possible or desirable between environments. Depending on the difference between the functionality available within the participating environments the functionality of the object being migrated should be retained at the destination as completely as possible.

Since the transfer of functionality is usually limited by the differences between the participating environments, functionality will not always be transferable completely and in some cases migration may not be possible at all. For example objects that use asynchronous message passing can not be migrated into an environment with synchronous message passing without fundamental changes to the destination environment, which are ruled out by the first objective (see also chapter 4 page 203).

4. Maximum of Object-Technology

The HLM migration mechanism should maintain as much as possible of the object-oriented characteristics of a particular object to be migrated. Furthermore, following the rules of encapsulation, the object itself should be able to control as much of the migration process as possible. For example a migration should not be performed without the consent of the object being migrated.

Since functionality is given a higher design priority than object technology compromises to the object characteristics of the objects being migrated are more acceptable then changes to their semantics. The implication for the developer that uses the HLM migration mechanism is that objects which are intended to be migrated can not be designed with the full fidelity of the particular language in all cases but only with a compatible set of functionality depending on the set of concepts supported by the HLM migration mechanism (see also the following subchapter page 72).

5. Minimal Requirements

The design of the HLM migration mechanism and its prototypical implementation should be independent from any specific facilities of any particular environment. An actual implementation of the HLM migration mechanism is free to take advantage of existing local services as long as the interoperability of the mechanism across heterogeneous environments is not affected.

Examples of facilities that can be used for particular implementation of the HLM migration mechanism are reflection as well as incremental compilation. The HLM migration mechanism does not require either technique, but both may be helpful in the implementation of the mechanism in certain environments.

6. Use of existing Technologies

The HLM migration mechanism should neither reinvent existing technologies nor create new ones unless necessary for particular aspects of the HLM migration mechanism. The design and implementation of the HLM migration mechanism should rather focus on features unique to heterogeneous migration at the language level.

In some cases the implementation of certain features is postponed as the use of existing technology for a particular purpose is obvious or does not add significant benefits to heterogeneous migration. For example location independent references to remote objects are not implemented by the prototypical implementation of the HLM migration mechanism as several solutions already provide the respective facilities (see also chapter 4 page 158).

7. Minimal Implications for the Developer

The use of the HLM migration mechanism should yield minimal implications for application developers, in particular no new programming model should be required. Creating applications that use the HLM migration mechanism should be possible with as few implications for the design of applications as possible.

Due to the higher priority of the above objectives the designer is not completely unaffected by migration. An application that takes advantage of the HLM migration mechanism will have to be designed with migration in mind at least for the objects that are intended to be migrated. In the context of the HLM migration mechanism migration is not transparent to the developer but can rather be controlled by the developer (see also chapter 4 page 156).

In addition to the design goals some explicit "non goals" of the HLM migration mechanism can be stated as well. As the least important aspect of the design, the compatibility of the HLM migration mechanism with other migration mechanisms has not been considered. Since the HLM migration mechanism tries to break new ground, the design should not depend on existing approaches and should only be influenced by these on a conceptual level. Beyond that, no comparable mechanism could be found with which interoperability would be possible.

The performance of the prototypical implementation of the HLM migration mechanism is also almost never considered important, except where performance is improved without implications for other design aspects. Since the HLM migration mechanism is intended as an platform for further experimentation, not as a production system, performance has been given a low priority.

3.1.2 Supported Concepts

The set of concepts that are supported by the HLM migration mechanism has been derived directly or indirectly in the form of further design considerations from the objectives established in the previous subchapter. The support for specific concepts often implies the rejection of alternative concepts. The following list describes those concepts that are supported as well as reasons why alternative concepts have been excluded.

- **Type Annotations**

  A fundamental requirement of the HLM migration mechanism is the use of type annotations by the participating language environments for object definitions, their components, method results, parameters, and variables. The type annotations are used to match the functionality required by the object to be migrated with the functionality available within a particular destination environment.

  The requirement for type annotations does not strictly imply the use of strong typing by the participating environments. The use of strong typing by language environments will nevertheless increase the applicability of the HLM migration mechanism. Untyped languages can be supported in principle through the use of type derivation [PaS1994] although that possibility is not further investigated here (see also chapter 4 page 189).

  In order to avoid confusion through name-conflicts among different languages the term *interface* is used synonymously for the type of an object as well as for its object definition. This convention is upheld regardless whether objects are defined via classes [GoR1983] or prototypes [UnS1987] or whether object definitions are implemented as objects themselves or only accessible as source files (see also chapter 4 page 192).

- **Single Inheritance**

  The HLM migration mechanism does only support single inheritance between object definitions in order to reduce the overall complexity. While languages that feature multiple

inheritance or delegation may implement the migration mechanism as well only objects that are confined to single inheritance will be able to be migrated.

The restriction toward single inheritance reduces the complexity of matching object definitions to an unambiguous case that can be applied universally across all language environments. Although equality of multiple inheritance or delegation schemes may eventually be proven, this additional complexity will not be considered for the HLM migration mechanism (see also chapter 4 page 192)

- **Unshared Local State**

  The objects to be migrated using the HLM migration mechanism are confined to unshared local state and message passing is required to access the state. This requirement essentially prohibits any sharing of state between objects as well as among objects of the same interface.

  The design of the objects to be migrated is affected by this requirement but the complexity of the HLM migration mechanism is also reduced significantly. The universality of method invocation is used in combination with the concept of single inheritance to overcome the differences of state access and protection mechanisms.

  The access to the state of objects is implemented differently by various language environments. For example some languages like in Smalltalk [GoR1983] distinguish between class and instance variables. Almost arbitrary schemes of so called *visibility* and *protection* of state are used among object-based languages. In order to avoid the complexity of matching different state access schemes only state that is local to an object to be migrated and not shared with other objects by means other then message passing is supported by the HLM migration mechanism.

  As a consequence implicit sharing among objects in the form of for example class variables in Smalltalk [GoR1983], protected members or friends in C++ [Str1991] can not be used by objects that are to be migrated. Since the notion of state that is shared among all instances of a class can not be easily maintained in the context of distribution as the corresponding class object of the source does not exist at the destination, the impediments implied by this restriction are just a natural consequence of migration.

  The only alternatives would be a distributed access to the shared state or a replication of the shared state. For example, JavaParty [PhZ1997] enables instances to access their classes in the context of distribution through remote method invocation. Unfortunately, both alternatives create residual dependencies between the participating environments.

  This restriction may seem inconvenient but does not imply severe impediments for a developer of applications. The necessary methods to access the components of objects that define the local state can easily be implemented and can eventually be generated automatically (see page 121). Some environments like CLOS [Ste1990] already use a similar convention in the form of so called accessor methods. The sharing of state can be implemented through dedicated container objects that are references by all objects that take part in the sharing.

- **Synchronous Message-Passing**

  The HLM migration mechanism supports synchronous message-passing and well known constructs as the only means to control the flow of execution among objects[10]. Neither generalized invocation mechanisms like generic functions as in CLOS [Ste1990] nor generalized constructs like block-objects as in Smalltalk [GoR1983] are supported. Only local method invocation and the constructs `if then else` and `while do end` Are supported by the HLM migration mechanism.

  The restriction to synchronous message passing does rule out the support of asynchronous method invocations. While the HLM migration mechanism may in principle also be applied

[10] A number of additional restrictions exist for the definition of method that are discussed in chapter 4 (see page 192).

to active environments, dealing with asynchronous message-passing requires a more intense control of the migration process. A single thread of control per environment is therefore assumed in the context of the HLM migration mechanism.

- **Inactive Objects**

  Due to the first objective that calls for "no changes to existing language environments" the migration of computations is not supported by the HLM migration mechanism. Language environments intended to be able to determine or recreate the executable state of objects to be migrated, would have to be modified extensively (see also chapter 4 page 179).

  As a consequence only inactive objects i.e. such that do not participate in any computations at the time of migration can be migrated. Migration requests for objects that are participating in computations, i.e. that have methods invoked while being subject to migration will be rejected. Only objects that are not referenced by activation records will be migrated by the HLM migration mechanism. Obviously, active objects, i.e. those that employ their own computations in the form of threads are also not supported.

  While this restriction might seem prohibitive at first it only confines the mechanism to object migration without thread migration. To fulfill this restriction some sort of liveliness analysis is required prior to the actual migration in order to ensure that only inactive objects are migrated.

All other concepts not described above are not supported by the HLM migration mechanism. Any concept can be used in applications that use the HLM migration mechanism but not for objects that are supposed to be migrated. The HLM migration mechanism will work only for objects that are confined to the supported concepts described above.

The chosen concepts do not limit the set of language environments that can participate in object migration unduly. Most existing object based language environments support the concepts described and the design of applications within these limitations should not restrict the use of the HLM migration mechanism significantly.

Appropriate tools can be developed that ensure for language environments that offer additional concepts that objects intended to be migrated obey these restrictions (see also page 121). The addition of support for further concepts in future versions of the HLM migration mechanism is possible and discussed in more detail in chapter four (see also page 189).

3.1.3 Prerequisites

Apart from the restriction to a number of language concepts some additional prerequisites have to be fulfilled by the participating environments and the objects to be migrated in order to implement the HLM migration mechanism. In some cases the respective features can be implemented without changes to existing environments if they are not already available.

In addition to the prerequisites stated for any migration mechanism in chapter two (see page 40) the HLM migration mechanism requires that participating environments are able to create new objects at runtime from object definitions added at runtime, which are based on globally unique type names.

- **Dynamic Loading and Binding**

  Due to the fact that the semantics of objects will be migrated by the HLM migration mechanism some sort of dynamic loading and binding will be required for the participating destination environments. The definitions of objects that are transferred as part of a migration have to be made available by the destination environment at runtime in order to make the newly added functionality accessible to the migrated objects. This has to be done not only in the sense of a late binding of a dynamic dispatch but in the form that new code can be added to a running program.

  In some environments the ability to add behavior at runtime is also called *dynamic compilation* and the ability to add parts of object definitions independently is called *incremental compilation*. In the context of the HLM migration mechanism the compilation of object definitions does not need to be incremental in the sense that individual method

definitions can be compiled independently. Only the transfer of complete object definitions will be supported by the HLM migration mechanism.

The HLM migration mechanism does not require the compilation process to be an integral part of the language environment as for example dynamic compilation in Smalltalk [GoR1983]. The compilation of new functionality can be done through a separate process as well as in the form of cross-compilation. However, the destination environment must be able to make the resulting object code available within the environment at runtime.

- **Dynamic Object Creation**

  Language environments that implement the HLM migration mechanism need to be able to create new objects based on the object definitions added at runtime by using the value of a variable, for example a string value. The creation of objects based on programming language identifiers i.e. compiler symbols alone will not be sufficient.

  The name of the object definition of the object to be recreated may not be part of the source code of the destination environment at compile time as the object to be migrated will not be known prior to the actual migration. Nevertheless, the migrated objects have to be recreated at runtime within the destination environment.

  The generation of appropriate method-code that contains the corresponding object creation expression is not a substitute. Such a code will itself need to be made available through dynamic creation of a corresponding object. The only alternative would be the use of incremental compilation of appropriately generated methods for existing object definitions of the destination which is only supported by very few environments.

- **Globally Unique Type Names**

  In order to simplify the implementation of the HLM migration mechanism the global uniqueness of names of types or interfaces is assumed. If two environments use the same type name, the corresponding types are assumed to be identical. This assumption can be applied without loss of generality and can be relaxed in future version of the mechanism for example through a consistent renaming of types (see chapter 4 page 174).

  The general problem of type equality, i.e. if two independently implemented types employ identical semantics can not be answered at all because of the problem of computability. The only alternative would be a distributed repository of types which would essentially ensure global consistency of types as for example in CORBA [Sie1996]. The use of such a repository does not add significant value to heterogeneous language migration and can be regarded as a detail of the implementation.

In contrast to the prerequisites stated above, some non-prerequisites can be stated as well. The HLM migration mechanism does not require runtime type information or reflection to be available within participating environments. Type information can also be retrieved from static object definitions like source files. The availability of reflective capabilities can be useful for an implementation of the HLM migration mechanism (see chapter 4 page 206).

Dynamic message passing, i.e. the construction of messages at runtime rather then compile time is also not required. All messages that are needed to implement the mechanism can be defined statically. Dynamic message passing can enable environments that do not offer dynamic object creation to participate in object migration if the necessary creation messages can be constructed at runtime.

Location independence is also not required by the HLM migration mechanism as only very few language environments support it. Various add-on implementations of location independence are available for existing language environments including heterogeneous ones [Sie1996]. While the availability of location independence enables different forms of migration (see chapter 4 page 158), it does not add significant value with regard to heterogeneity.

3.2 Architecture

Without loss of generality a migration of objects among disparate environments can be simplified to the “one object and two systems” case were a single object is migrated from a

source to a destination environment. As no detailed knowledge about the destination is assumed, a migration is also regarded as independent of previous or further migrations of other objects to the destination environment and independent of whether the object originated from the source environment or was migrated to it.

The migration of a single object does also generalize the migration of a set of objects that are migrated consecutively one by one. However, the migration of a single object will not permit optimizations possible for the migration of sets of similar objects that do not need to be migrated independently (see also chapter 4 page 152).

The following sections and subchapters describe the architecture of the HLM migration mechanism in the context of the migration of a single object. The architecture can be applied to the consecutive migration of sets of independent objects as well as the collective migration of sets of similar objects. The architecture is introduced by a stepwise description using progressive disclosure of the various design issues involved.

Migration Control

Following the principles of object technology, especially encapsulation, a naive approach to migration would probably attempt to implement a migration mechanism for language objects as part of the behavior of the object to be migrated itself. Such a straight-forward approach is not advisable if language migration is to be performed with only minimal requirements at both source and destination environments.

An implementation of the migration mechanism as part of the behavior of the object to be migrated, implies that the migration mechanism itself is also migrated during migration. This increases the complexity of the object to be migrated unnecessarily especially if the management of communication-failures also has to be implemented as part of the mechanism.

As the environments involved may fail during migration or the communication between the systems may become unstable, some coordination between the source and the destination environment is necessary in order to provide recovery from communication failures. Implementing the necessary communication mechanism within the object to be migrated would imply an undue overhead.

Placing a generic migration mechanism within another entity will ease the implementation of the migration mechanism and will also simplify the coordination of the necessary communication. The source as well as the destination environment may serve as alternative candidates for the implementation of the migration mechanism.

A reasonable design rationale minimizes the requirements of the migration mechanism for the behavior of the object to be migrated. The self-determination of the objects to be migrated does not have to be compromised as long as the objects are involved in every step of the migration process and can raise a “veto” for every critical decision.

Given that three software entities participate in a migration, the question arises, which entity should control the overall migration process ? There are obviously also three answers:

- **Object Control**

  From the standpoint of encapsulation the obvious answer is: the object to be migrated should control the migration process. Unfortunately control of the whole migration process by the unit of migration requires the availability of the object being migrated at all times during the migration process. While feasible in principle optimizations for large sets of uniform objects that benefit from being transferred collectively are precluded.

  As the object to be migrated has no knowledge about the destination environment and probably only limited knowledge about the source environment, implementing the control of the migration process as part of the object to be migrated would require large extensions to its behavior.

- **Destination Control**

  An alternative for the control of the migration process is to place the responsibility within an entity of the destination environment. This makes sense from a transactional standpoint, as the destination environment is supposed to complete the migration by making the migrated object available. Unfortunately destination control implies a high communication overhead.

  As the destination environment has no knowledge about the object being transferred and may even differ significantly from the source environment, a destination entity is not well positioned to determine what has to be transferred during migration. All necessary information will have to be brought to the destination first. The entity of the destination will have to "understand" the characteristics of the object to be migrated and eventually of the source environment before the migration may actually be performed.

- **Source Control**

  Lastly, an entity of the source environment may control the migration process. The source environment provides all available information about the object to be migrated and an entity of the source will therefore be able to decide what needs to be transferred during migration with less communication overhead.

  As a source entity does not have any knowledge about the destination environment the necessary information will have to be transferred to the source from the destination. A controlling entity of the source will have all the available information about the object to be migrated but will have to query the destination environment in order to find out if and how the object can be migrated.

It is quite obvious that none of the three entities is ideally positioned to control the migration process and none will be able to do so without the cooperation of the other two. While the amount of information to be transferred during the migration of an object itself can be regarded as independent of the actual controlling unit, this will not be the case for the communication overhead for the decision making of the migration mechanism.

The main objective for the migration mechanism is to make the object being transferred operable at the destination. All the available knowledge about the object to be migrated resides at the source. It appears to be easier to transfer the conditions of the destination to the source and determine at the source whether migration is possible and what needs to be migrated.

Alternatively, transfer of all relevant information to the destination appears to be more wasteful and impractical. It will also be much more difficult to let the object to be migrated take part in the decision whether and how it should be migrated as it will still be part of the source environment when this decision has to be made.

The simplest form of migration would transfers a representation of an object to the destination environment which is then in charge of implementing and recreating the object. Although quite simple this approach has some severe disadvantages. The destination environment relies solely on this representation, decreasing the probability of migration-success. The source environment would therefore be forced to send as much information as possible..

Alternatively, as the source environment has no knowledge about the destination environment it is necessary to negotiate whether and under what circumstances the destination is able to implement the object to be migrated prior to the migration. The source environment is more apt to conduct this negotiation as it has potentially complete knowledge about the current implementation of the objects in question. This approach also potentially allows the object being migrated to participate in the negotiation process (see also chapter 4 page 177).

As the object itself and a destination entity appear to be less suited for the control of the migration process source control is chosen for the HLM migration mechanism. This will minimize the overall communication and the amount of work performed where a migration proves to be impossible.

The HLM migration mechanism uses one object called *Porter* for each participating language environment to perform and control the migration of objects. The Porter objects communicate

with each other and exchange the information that is necessary to process the migration. The Porter objects essentially implement the migration mechanism as part of their behavior.

Only the absolutely necessary parts of the HLM migration mechanisms will be made part of the behavior of the objects to be migrated. In the context of the HLM migration mechanism the objects to be migrated will be called *Migrateable* objects. Figure 3.a shows a preliminary version of the architecture of the HLM migration mechanism.

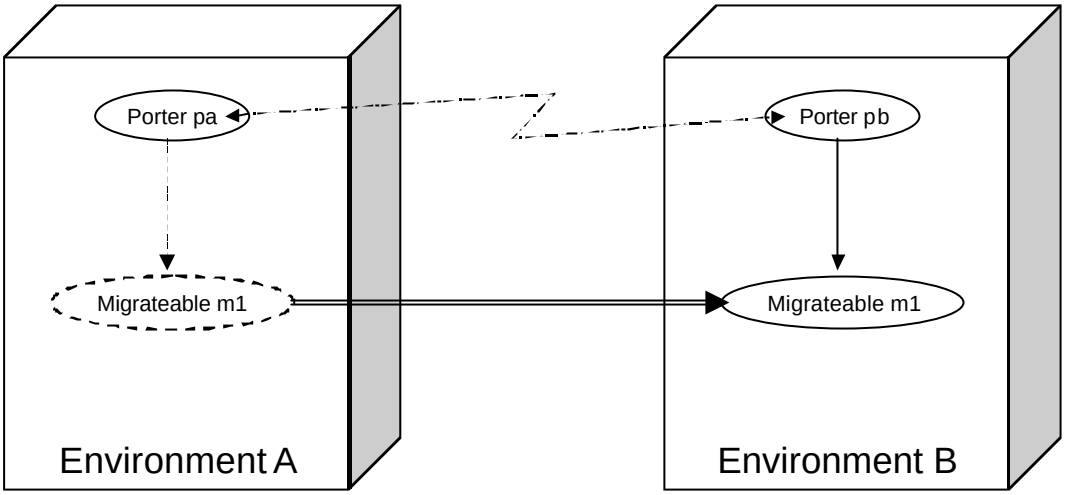

Figure 3.a: The first two elements of the architecture of the HLM migration mechanism are the Porter and the Migrateable objects (shown here as ellipses). In the above example Porter objects pa and pb communicate in order to control the migration of the Migrateable m1 and to transfer the necessary information.

The alternative to let the Porter objects handle the migration process completely would violate the objective that a maximum of object technology has to be used. The migration process should be as much self-controlled as possible by the objects to be migrated and should require as few additions as necessary to the behavior of these objects. The HLM migration mechanism defines a message protocol for the Migrateable objects in order to control the decision whether such an object is willing to be migrated by the Porter or not.

Implementing only the minimal part of the migration mechanism within the behavior of the Migrateable objects will also limit the changes that are necessary to the design of objects in order to render them migrateable. It will also reduce the complexity of the HLM migration mechanism itself as less dependencies exist.

This decision does also conform with the first objective for the HLM migration mechanism, which requires that definitions of the participating languages should not be changed. The capability of objects to migrate is associated with their type and not implemented orthogonal to their type as for example within Emerald [Jul1993]. The later would only be possible through changes to the participating language environments.

Relationship Management

The definition of Porter and Migrateable objects alone will not be sufficient to implement object migration in the general case. Objects that are supposed to be migrated are in most cases related to other objects they need to exchange messages with. Various kinds of relationships exist that need to be taken care of during the migration process.

A relationship between objects is made explicit through a unidirectional reference between the objects. An object that is referenced by the object to be migrated or references itself the object to be migrated is called a *related object*. The set of related objects of the object to be migrated has to be determined at the time of migration because the references between the objects can change at any time prior to migration. Figure 3.b illustrates the discriminating cases of relationships between objects.

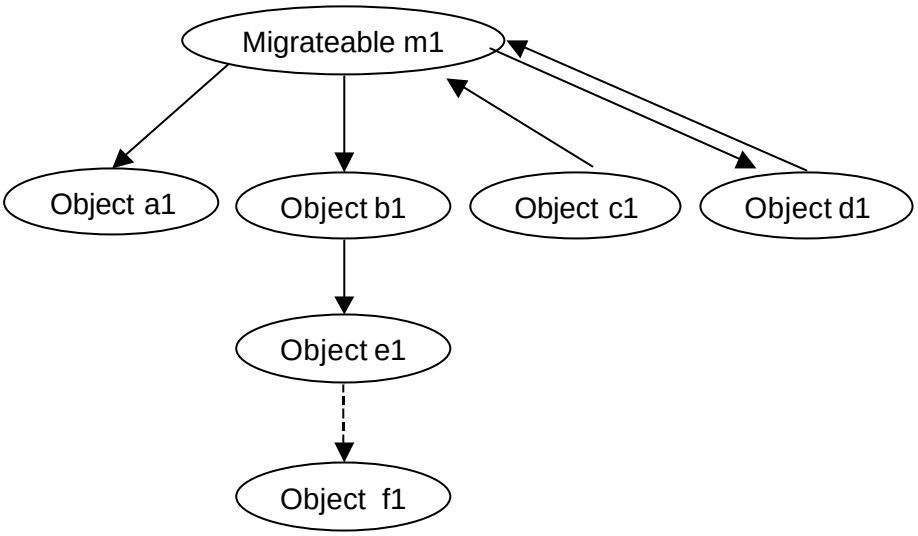

Figure 3.b: A Migrateable m1 may hold references to other objects (a1, b1 and d1 in the above example). The Migrateable m1 may be referenced by other objects (c1 and d1 in the above example) as well. Other objects can be indirectly affected (e1 and f1 in the above example).

The dfferent relationships that may exist between a Migrateable and other objects can be identified using the example in figure 3.b as follows:

- **explicit**

  If a Migrateable references another object, for example m1 references object a1, then a1 is called to be *explicitly related* to m1.

- **indirect**

  If an object explicitly related to a Migrateable references another object, for example m1 references object b1 and object b1 references object e1, then e1 is called to be *indirectly related* to m1.

- **transitive**

  If an object indirectly related to a Migrateable references another object either explicitly or transitively, for example m1 references object b1 and object b1 references object e1 which references object f1, then f1 is called to be *transitively related* to m1.

- **implicit**

  If a Migrateable is referenced by another object, for example object c1 references m1, then c1 is called to be *implicitly related* to m1.

- **mutual**

  If both a Migrateable and a related object reference each other, for example object d1 references m1 and vice versa, then d1 is called to be *mutually related* to m1.

How the various relationships are treated during the migration process depends on whether the references between the objects can be maintained in the context of distribution. The relationships between objects are independent from migration if the existing references are not invalidated through a migration and messages can be send between the objects even after migration. Any object can then be migrated freely.

In order to achieve this freedom, transparency of location and of invocation have to be available within all participating environments. Unfortunately, only few language environments offer this level of transparency as a built-in feature. Several technologies to achieve this level of transparency are available even for the heterogeneous case [Sie1996].

Rather then integrating an existing transparency technology the design of the HLM migration mechanism will focus on the more interesting question what has to be done if transparency of

location and invocation are not available, which is also the default case among existing language environments.

If the references between an object to be migrated and its related objects can not be maintained in the context of distribution the related objects have to be migrated as well or the relationships have to be cut. This can only be decided upon and performed in cooperation with both objects. Otherwise an invalid reference will exist if the object being migrated is simply deleted in environments that support manual memory management. Within environments with garbage collection an object to be migrated would not be collected after it has actually been migrated.

Implicitly related objects are especially problematic in this regard. A Migrateable is not aware of its implicit relationships and a reference from an implicitly related object to a Migrateable will be broken by the migration process without prior notification. This will probably have disastrous consequences. The HLM migration mechanism needs to address implicit relationships between Migrateables and other objects in one of the following three ways:

- Recursive Traversal

  The obvious and straightforward way to handle implicit relationships is a traversal of all relevant references at the time of migration. Since the objects that may reference the object to be migrated are unknown this task is equivalent to a full traversal of all objects within an environment similar to garbage collection mechanisms.

- Central Registration

  An alternative way to handle implicit relationships may be implemented by registering the implicitly related objects with the special object of the respective environment, for example, the Porter object. As many Migrateable objects may exist in an application but only relative few will actually be migrated the Porter object will then have to manage a lot of implicitly related objects in vain.

- Decentral Registration

  As a third option only mutual relationship may be allowed between the Migrateable object and its related objects. The designer of an application will be required to match each implicit relationship with an explicit one. In order to make this task as easy as possible each Migrateable object can offer a way to register its related objects.

A recursive traversal can not be implemented without changes to language environments and will probably impede the performance of the migration mechanism prohibitively. A central registration also has severe performance implication as every creation or destruction of a reference has to be matched with a registration or deregistration of the related pair of objects.

Due to this consideration decentral registration is chosen for the HLM migration mechanism. This alternative provides better control for the Migrateable objects with regard to the determination of related objects. Migrateable and related objects are effectively required to be designed to use only mutual relationships. This design constraint for applications that use the HLM migration mechanism can be supported by appropriate development tools (see page 121).

Objects related to a Migrateable will either be migrateable themselves or not. Objects that can not be migrated but are related to a Migrateable object still need to use mutual relationships with the respective Migrateable. These objects will be called *Owner* objects, as they own a reference to a Migrateable object. Owner objects effectively define the border of the set of objects that can be migrated in the lattice of all objects of an application.

The designer has to decide which objects of an application are Migrateable objects, which have to be Owner objects and which are not related to Migrateable objects and can be implemented independently. Appropriate tools can help with these design decisions as well as to generate the necessary object definitions (see also page 121).

Depending on the individual application some Migrateable objects will need to access objects of a particular language environment that are predefined and can not be changed by the designer of the application. For example input/output related objects fall into this category. Such objects

have to be encapsulated in Owner objects that are able to handle a possible migration of the related Migrateable objects appropriately.

An number of standard objects as well as special object called *Environment* will be required for each implementation of the HLM migration mechanism in order to provide access to basic services of every participating language environment. The Environment object will also help to ease the design of applications of the HLM migration mechanism. There should be only one Environment object per participating environment that will also be known to the Porter object.

All other existing objects that can not be extended for mutual relationships will have to be separated by the designer of an application from the Migrateable objects through the use of Owner objects that will maintain the implicit relationships on behalf of the Migrateables. Figure 3.c illustrates the various relationships of an object that may exist within the context of the HLM migration mechanism.

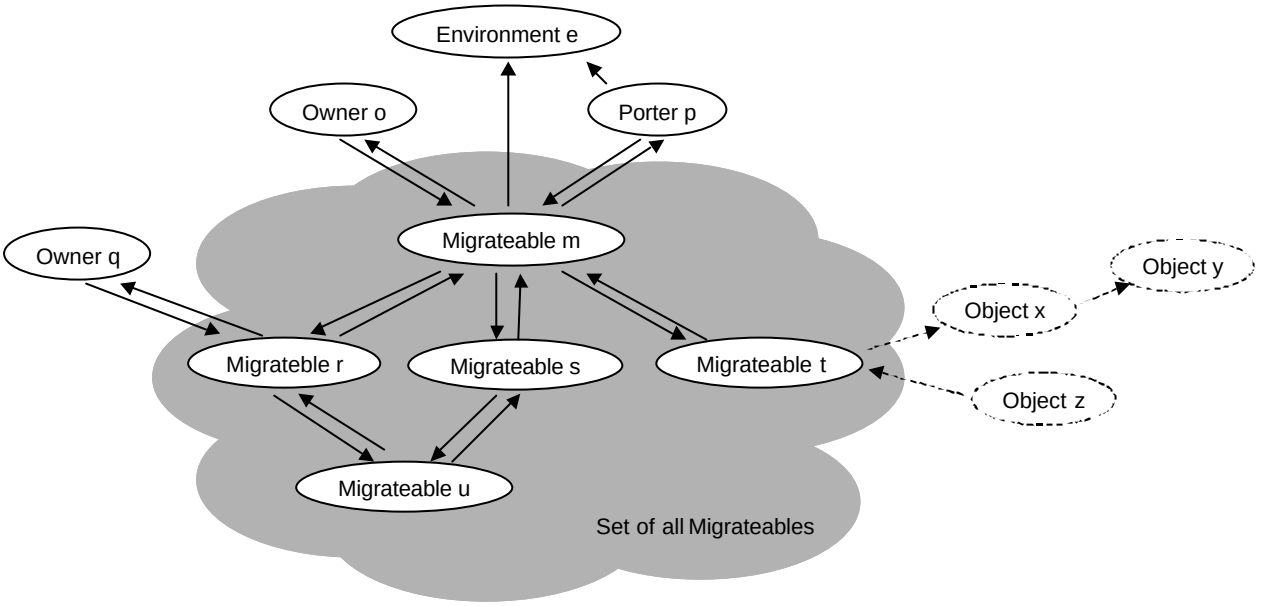

Figure 3.c: The relationships between objects (shown here as ellipses) have to be designed differently in the context of the HLM migration mechanism. In the above example the Migrateable object m is mutually related to other objects that are either migrateable themselves like the objects r, s and t or Owner objects like the objects o and q or the Porter object p. Neither explicit nor implicit references between Migrateable and other objects are allowed (shown as dashed lines between Migrateable t and the objects x and z).

Objects that are not Owner objects but reference a Migrateable object and do not register with it are not supported (object z in the example of figure 3.c). Some environments as for example Loops [SB+1983] offer features to manage such references as well as to conduct recursive traversals to detect such references but such means are not commonplace among existing language environments.

No prohibitive actions will be implemented as part of the HLM migration mechanism in general. There may be cases where such references are necessary and the designer of the respective objects will be responsible for their management. Appropriate design tools can help in identifying and managing these cases (see also page 121).

Migration Architecture

The reference architecture of the HLM migration mechanism also called the *migration architecture* consists of four kinds of objects: a Porter object that controls the migration, Migrateable objects that can be migrated, Owner objects that are related to the Migrateables but are not migrateable themselves and an Environment object that will make services of the

particular language environment accessible to the Migrateable objects. Figure 3.d illustrates the migration architecture.

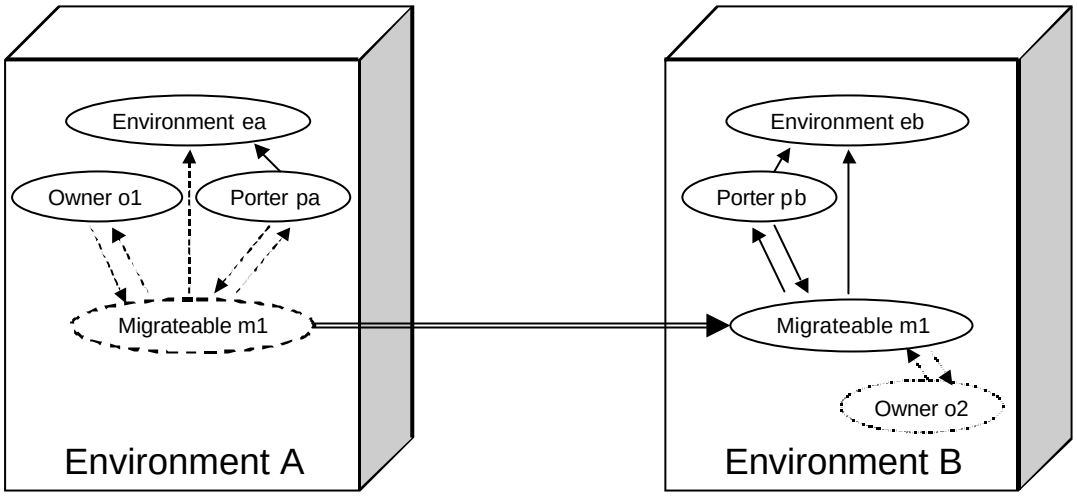

Figure 3.d: The migration architecture of the HLM migration mechanism. Both environments employ their own Porter object pa and pb as well as Environment objects ea and eb respectively. The Migrateable object m1 can be dsconnected from Owner object o1 in environment A, migrated to environment B and with the help of the Porter object pb may be connected with a completely different Owner object o2.

The various object roles that are identified in the migration architecture are formalized via object definitions that specify the corresponding behavior. These have to be used in their language specific variants for all objects participating in the HLM migration mechanism. The object definitions are related through a single inheritance relationship. An object definition called for the root of the resulting inheritance hierarchy named Object is added as well. Figure 3.e illustrates the inheritance tree of the migration architecture.

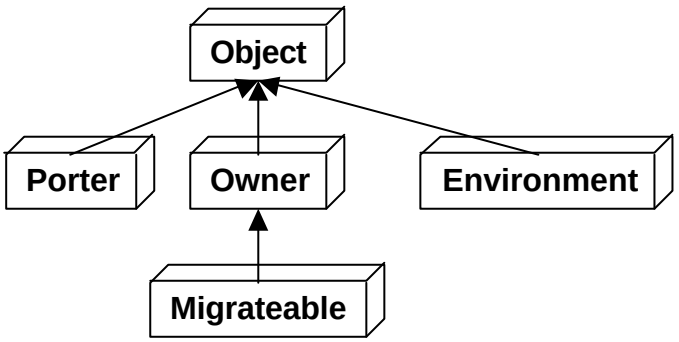

Figure 3.e: The migration architecture of the HLM migration mechanism is constructed of object definitions that are related by a single inheritance relationship and form a inheritance hierarchy.

The inheritance hierarchy of the migration architecture is quite simple. The Migrateable object definition inherits part of its behavior from the Owner object definition. All other objects of the architecture inherit from the object definition Object. The behavior of the object definitions within the architecture of the migration mechanism is described in the following subchapters.

3.2.1 Porter

The Porter object is intended to coordinate the migration process. It serves as the addressee of migration requests and initiates and controls all steps of the migration process. It essentially implements the HLM migration algorithm through its behavior and corresponds with the Migrateable objects, Owner objects and the Environment object in order to perform a migration.

Although only one Porter object is necessary to perform a migration, an actual implementation may choose to employ several Porter objects for example in order to allow several migrations

simultaneously. To do so the respective environments need to be multithreaded and synchronization mechanisms like Monitors [Hoa1974] will have to be used for all objects that can potentially be shared by different migrations (see also chapter 4 page 203).

The prototypical implementation of the HLM migration mechanism does not strictly require the participating environments to be single-threaded but it is confined to one Porter object per environment and only one migration at a time may take place. If the HLM migration mechanism is implemented on top of a multithreaded environment the designer of applications will have to provide the necessary synchronization manually.

3.2.2 Owner

An Owner object is an object that is not migrateable but references one or more Migrateable objects. Owner objects correspond with Migrateable objects in order to establish a mutual relationship and to deconstruct it once the Migrateable object is actually migrated. Owner objects effectively define the boundary of the set of Migrateable objects within the set of all objects of an application.

Owner objects participate actively in the migration process. They take part in the decision whether the migration of the Migrateable objects takes place and they cooperate in the actual migration. They invalidate their references deliberately as the respective Migrateable objects prepare to be deleted from the source environment.

Owner objects of the destination environment can actively participate in the migration process as well. New relationships between Owners objects and Migrateable objects that have just been migrated into an environment can be established with the help of the Porter object. However their actual role depends of the particular application that uses the HLM migration mechanism.

3.2.3 Migrateable

A Migrateable object is an object that is able to be migrated. It needs to be aware of all its relationships with other objects of the source environment. A registry of all related owner objects is managed by an Migrateable object. As the object definition of the Migrateable objects is derived from the object definition of the Owner objects, a Migrateable can fulfill the role of an owner of other Migrateable objects as well (see also page 96).

The designer of an application of the HLM migration mechanism has to decide which objects have to be migrateable. These objects will have to inherit from the Migrateable object definition. For objects that will remain within the source environment but are related to Migrateable objects, inheritance from the Owner object definition will have to be used instead.

The Migrateable object has to give its consent whether it can be migrated or not. It will make that decision based on its current state and will query all its Owner objects whether they agree as well. If only a single Owner disagrees with the requested migration the migration will have to be aborted (see also page 96).

3.2.4 Environment

For migrated objects to be useful within destination environments they need to correspond with built-in objects of that environment. These will be made available through the use of an Environment object that serves as a pathway for messages between the objects involved. The relationship of a Migrateable object with the Environment object is established via the Porter of the destination environment.

Although several Environment objects could be used within a particular destination environment, all of these would in principle serve the same role. Therefore only one Environment object per participating environment is used by the HLM migration mechanism. The implementation of the Environment object may differ significantly among environments.

Environment objects are referenced by Migrateable objects but are not involved in the decision making of the migration process. The Migrateable object initiates any housekeeping that has to be performed by the Environment object prior to migration. The Porter will inform the Environment object which migrated objects are no longer available and will trigger any housekeeping to be executed by the Environment object after migration.

3.2.5 Object

An universal object definition named Object is used as the root for all object definitions including those for the Porter, Owner, Migrateable and Environment objects. Despite the fact that all other objects do not reference a Migrateable object they are not completely independent of the whole migration process.

The object definition of Object contains a minimal behavior that is required for all objects on the behalf of the HLM migration mechanism. This includes a test whether an object is a Migrateable object or not and a query of the name of the object definition of an object. Depending on the support available within a particular environment this requirement may be lifted for the individual case as other means to distinguish Migrateable objects may exist.

3.3 Abstractions

The HLM migration mechanism needs to transfer all information about an object to be migrated in such a way that the object can be recreated and is able to function as intended within the destination environment. The information to be migrated include at least the functionality and the state of the object to migrated.

In the context of heterogeneity different representations can be used for various information that have to be transferred as part of the migration of objects. A definition of an object to be migrated may either be generated in the format of the source and then parsed with an additional parser at the destination. Alternatively it may be generated with an additional generator at the source in the format of the destination and then parsed natively at the destination. If all participating environments either implement additional parsers or all implement additional generators, arbitrary migration will be possible.

To allow migration between an arbitrary number of languages every participating environment would have to either parse or generate every other object definitions in a brute force approach. Given that n environments can potentially participate in migration, every environment would have to implement n-1 parsers for all other representations except its own, resulting in a total of n^2-n parsers to be written. The same number of generators would have to be written otherwise.

Alternatively a single representation can be used. Any representation of the participating environments could be used but this would set an undue prejudice. A single generator and a single parser will be necessary for each environment if the common representation is artificial and not native to any of the environments. The use of a common representation reduces the amount of work to be done to 2*n generators and parsers to be written altogether . Figure 3.f provides a principal overview of the alternative approaches.

In the context of heterogeneous language environments a great variety of source, intermediate, virtual and binary formats are available and can be used for both the representation of the state and the behavior of the objects being transferred. Unfortunately not all of these representations are used by all existing language environments. Hardware-oriented binary representations are not compatible with virtual machine implementations and vice versa. Intermediate representations are only available within few language environments. The only format that is available for all language environments is source code.

Although binary, virtual or intermediate formats may offer performance advantages as they can be translated more directly the source format was chosen for the HLM migration mechanism, as it also eases the implementation and debugging of both the prototypical implementation and the objects to be migrated. The use of other forms of representation can be investigated in future versions of the HLM migration mechanism (see also chapter 4 page 179).

As part of the HLM migration mechanism a source code representation of the semantics of the object to be migrated at the destination is generated within the source environment and transferred to the destination environment. The state of the objects to be migrated is also transferred in source code format. A binary representation of the state of objects would require less communication overhead but a simple source code format was chosen for ease of implementation and debugging.

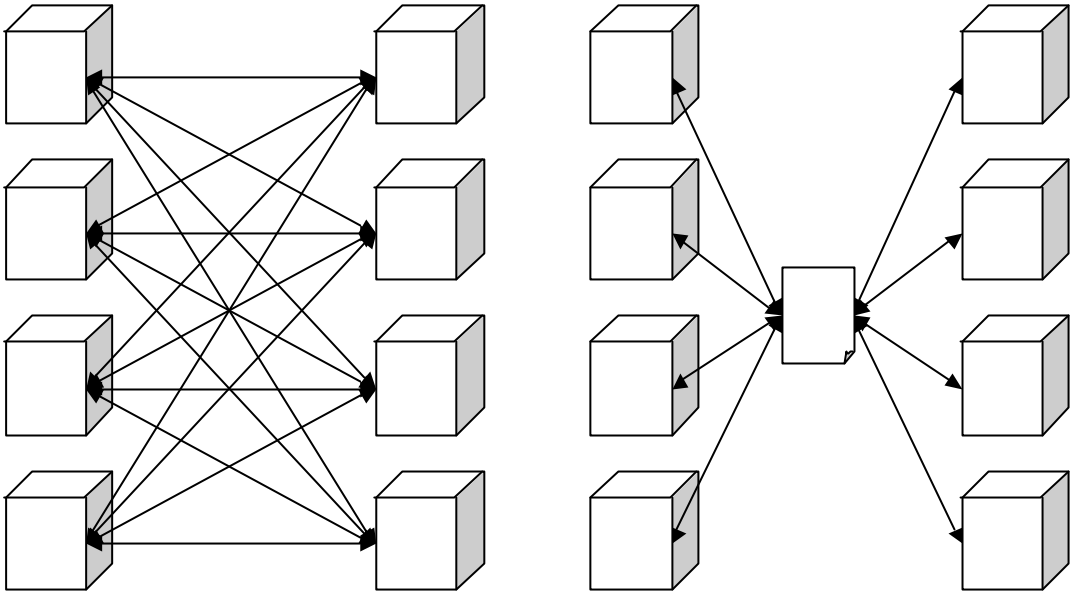

Figure 3.f: The use of a common representation reduces the number of translations required for n supported environments from n²-n to 2*n. Each participating environment requires one translation from the common representation into the native representation as well as one with additional checks in the opposite direction.

The HLM migration mechanism abstracts from the objects to be migrated through the use of so called *interfaces*. An interface captures the definition of an object independent from its actual state. The state of an object is captured as a so called *object representation*. The interface and the representation of an object are established through abstraction languages.

The abstraction languages used by the HLM migration mechanism have been specifically designed for the mechanism. The language used to represent the semantics of objects is called *Generalized Object Abstraction Language* (GOAL) and the language use to represent the state of objects is called *Object Representation Language* (ORL) (see also page 88).

The GOAL language consists of two parts, one used for interface declarations and one used for interface definitions with the latter being a syntactical extension of the first. Both the interface declaration and the interface definition are also synonymously called the *interface* of an object. The term interface definition is equivalent to the term object definition and is used to denote the GOAL abstraction of an object of any particular language environment.

The GOAL language is based on a combination of a subset of the Interface Definition Language (IDL) defined by the Object Management Group (OMG) [Sie1996] and of a subset of the Java programming language defined by Sun Microsystems [GJS1996]. The HLM migration mechanism does not rely on either the OMG CORBA standard or the Java language in general. This combination was chosen for ease of implementation and for reasons of extensibility. A completely different language was used in an earlier version of the prototypical implementation.

For the representation of the state of the objects to be migrated a language called Object Representation Language (ORL) is used. The ORL language is based on a syntax similar to the functional programming language Lisp [Ste1990] and is able to express atomic data values as well as data structures of arbitrary complexity.

3.3.1 Interface Declarations

The *interface declaration* part of the GOAL language is used to specify the behavior of the objects and optionally the structure of the state of objects. The behavior is described in terms of the signatures of messages that can be sent to an object. A signature is used for every method an object implements including the accessor methods used to manage its state.

A GOAL interface declaration contains all the necessary parts of the interface of an object that are required by the HML1 migration mechanism to process a migration. In addition the

components of an object, i.e. the structure of the state of an objects can optionally be part of its interface declaration. This optional information may be required by future versions of the migration mechanism (see chapter 4 page 156).

An interface declaration in GOAL is composed of an interface name, optionally the name of an *ancestor interface* the one being declared inherits from, an optional set of components and a set of signatures. The components of an interface that define the structure of the corresponding objects are defined as pairs of interface and component names. The signatures that describe the messages that can be sent to the corresponding objects are defined in terms of the interface name of the result, a signature name, and a list of parameters consisting of pairs of interface and parameter names . Figure 3.a shows an example of a GOAL interface declaration.

```
interface Personal_Account : Bank_Account {
Bank_Customer customer;
Bank_Manager manager;
Currency balance;
Bank_Account open_account(
     Bank_Customer c,
     Bank_Manager m,
     Currency b);
Bank_Account assign_manager(Bank_Manager m);
Bank_Manager get_manager();
Currency get_balance();
Bank_Account deposit(Currency c);
Bank_Account withdraw(Currency c);
Currency close_account();
};
```

Excerpt 3.a: An interface declaration expressed in the GOAL language. In the above example the interface `Personal_Account` inherits from the interface `Bank_Account` and contains the components `customer`, `manager` and `balance` of the interfaces `Bank_Customer`, `Bank_Manager` and `Currency` respectively. `Personal_Account` also defines signatures for opening an account, assigning and querying the manager and the balance of the account, depositing and withdrawing money and closing the account.

The interface declaration part of GOAL is essentially a subset of OMG-IDL, which is part of the Common Object Request Broker Architecture (CORBA) [Sie1996]. This representation was chosen because CORBA can potentially be used to implement remote method invocations across heterogeneous environments, a feature that may be added to the migration mechanism in the future (see also chapter 4 page 158).

The information provided by the interface declaration is sufficient for a comparison of the functionality available at the destination with the requirements of the object to be migrated (see also page 103). Alternative representation would either involve more details of implementation or have a more formal character.

Using a formal specification like representation would allow for additional information like pre- and post-conditions of method invocations. While some language like Eiffel [Mey1992] provide, for example assertions that allow the definition of such conditions, these are not supported by the majority of language environments. The use of such a representation would therefore limit the set of supported languages unnecessarily.

3.3.2 Interface Definitions

The interface definition part of the GOAL language is essentially an extension of the interface declaration part with signature bodies that are composed of variable definitions and executable statements. The variable definitions are composed of interface and variable names. The statements that can be used comprise assignments to local variables, message passing expressions, object creation expressions, the control structures `if-then-else` and `while-do-end` and `do-while` as well as `return` statement.

The components of an object are a required part of a GOAL interface definition and no longer optional as with interface declarations. The set of components consists of interface and component names that are listed prior to the set of signatures. Excerpt 3.b shows an example of a GOAL interface definition.

```
interface Personal_Account : Bank_Account {
Bank_Customer customer;
Bank_Manager manager;
Currency balance;
Bank_Account open_account( Bank_Customer c,
                        Bank_Manager m,
                        Currency b) {
     customer = c;
     manager = m;
     balance = b;
     return this;};
Bank_Account assign_manager(Bank_Manager m) {
     manager = m;
     return this;};
Bank_Manager get_manager() {
     return manager;};
Currency get_balance() {
     return balance;};
Bank_Account deposit(Currency c) {
     balance = balance.plus(c)
     return this;};
Bank_Account withdraw(Currency c) {
     balance = balance.minus(c);
     return this;};
Currency close_account(){
     Currency c;
     c = balance;
     balance = 0;
     manager = null;
     customer = null;
     return c;};
};
```

Excerpt 3.b: An interface definition expressed in the GOAL language. In the above example the method bodies are added to the signature of the interface `Personal_Account` introduced in excerpt 3.a. The methods essentially use assignment and return statements with the exception of the `deposit` and `withdraw` methods that use message passing expressions.

The interface definition part of the GOAL language is essentially a subset of the programming language Java [GJS1996]. Java was chosen because of the possibility to prove the executability of the represented object semantics easily. The HLM migration mechanism does not rely on Java in any form though Java can be a participating environment that implements the HLM migration mechanism.

The set of statements used by the GOAL interface definitions comprises only a small subset of the Java language. Only constructs that are found among most programming languages or can be translated easily into most programming languages are used. The executable part of the GOAL language has been kept as small as possible but is computationally complete.

Almost any other executable representation can be used and a completely new representation could be invented that requires its own interpreter or compiler. Although the latter approach was chosen with an earlier version of the HLM migration mechanism it proved to add little value to the general problem of heterogeneous migration.

Some other executable representation, for example Lisp would provide more flexibility through its interpreter based execution model. However, the general use of a compiler based execution

model seemed more appropriate as the majority of existing environments follow that model. The subset of the Java language was therefore chosen as the most adequate.

3.3.3 Object Representations

The object representation that is used to transfer the state of objects to be migrated is essentially an externalization or serialization of the objects in question. The HLM migration mechanism uses a separate language called Object Representation Language (ORL) for this purpose. The representation of the state of an object in ORL is composed of two parts, the interface and identity information part and the object initialization information as the second part.

The first part of an ORL representation consists of a list of pairs of interface-names and unique numbers called the *object migration identifier*, one pair for each object to be migrated. This format essentially associates every object to be migrated with an unique number in order to identify it with the corresponding newly created object within the destination environment in the context of the current migration.

The second part of the ORL representation consists of a list of pairs of a unique number identifying a particular object to be migrated and a list of either textual representations for atomic values, unique numbers preceded by a colon ":" sign that are used to reference other migrated objects, star signs "*" that indicate optional components, or all of the above in nested lists delimited by pairs of parentheses. This format is used to reconstruct the individual migrated objects within the destination environment. Excerpt 3.c shows an example of an ORL object representation of an object like the ones defined in excerpt 3.b above.

```
Personal_Account 1
Bank_Customer 2
Bank_Manager 3
Currency 4
|
1 (:2 :3 :4)
2 ("Max Mustermann")
3 ("Boris Banker")
4 ("Euro" 1000))
```

Excerpt 3.c: The ORL language can be used to represent the state of objects. In the above example a list of pairs of interface-names and numbers defines and identifies the objects to be migrated. This list ends with a single vertical bar "|". The following lines contain for each identified object a list of atomic values or identifications of referenced objects that comprises the state of the respective objects to be migrated.

The format of the GOAL object representation may seem arbitrary but is tightly related to the HLM migration mechanism. Although less compact than an binary representation a source code format was chosen for the representation of the state of objects to be migrated in order to ease debugging of the prototypical implementation of the HLM migration mechanism and of the objects being migrated. Future version of the HLM migration mechanism may nevertheless use different representations for performance reasons.

As an alternative, an existing serialization format like the externalization service of CORBA [Sie1996] could be used instead of a newly invented one. This new implementation was chosen because only a limited functionality was necessary for the externalized representation. Using a complex mechanism like the CORBA Externalization Service appeared to add little to the general problem of heterogeneous migration.

3.4 Standard Interfaces

The HLM migration mechanism is designed to work with only minimal knowledge about the destination environment prior to migration. Yet a designer of an application will need some common functionality in order to implement Migrateable objects that can be useful within the destination environment. A set of interfaces called *standard interfaces* is used to provide the necessary common functionality among environments.

The set of standard interfaces used for the HLM migration mechanism consist of two parts: an inheritance hierarchy of commonly used interfaces and a small set of so called singular interfaces that is not part of the inheritance hierarchy. The set of *singular interfaces* was chosen in order to represent abstractions for atomic values that occur in hybrid language environments. Atomic values that are not represented by objects will also be called *singular objects.*

Because of the use of singular interfaces the HLM migration mechanism is able to support both pure and hybrid language environments. Atomic values of hybrid languages can be abstracted through singular interfaces and recreated as objects of pure languages and vice versa. As a consequence the definition of subtypes of singular interfaces in pure environments is not supported by the HLM migration mechanism.

The standard interfaces have to be implemented in the context of the participating environments. A native implementation will not always be required though. A wrapper around existing object definitions that provides the necessary functionality will be sufficient in most cases. The singular interfaces may not be defined within hybrid environments at all as only the appropriate constructs of the destination need to be generated during the migration process.

The selected set of standard interfaces used in the prototypical implementation of the HLM migration mechanism serves as a starting point for further experimentation and is neither complete nor comprehensive. The set may well be changed or extended in future versions of the mechanism and in response to the support of additional language environments.

The actual set of standard interfaces that can be used for applications of the real world will have to be defined by standardization efforts rather than research. In order to support different application purposes several disjunctive or alternative sets of standard interfaces can be defined as well (see also chapter 4 page 154).

All standard interfaces of the prototypical implementation of the HLM migration mechanism bear names with the prefix `OM_` an abbreviation of “object migration”, in order to avoid name conflicts with existing interfaces of particular environments. Additional standard interfaces may follow this convention or use different techniques to avoid name conflicts.

The standard interfaces of the HLM migration mechanism include the singular interfaces `OM_Boolean`, `OM_Integer`, and `OM_Float`. The singular interfaces are only specified through interface declarations. The actual implementation of the standard interfaces is specific to each participating language environment. Some environments may chose to map them to built-in types while other may use wrapper code around existing object definitions. Regardless of their implementation singular objects will be captured with a common textual representation within ORL object representations.

The other standard interfaces comprise the basic interfaces `OM_Object`, OM_Character, `OM_String`,, the data-structure oriented interface `OM_Set`, the input/output oriented interfaces `OM_Stream`, `OM_File`, `OM_Directory` as well as the network oriented interfaces `OM_ServerSocket` and `OM_Socket`. The elements of the migration architecture are also defined as the standard interfaces `OM_Porter`, `OM_Owner`, `OM_Migrateable` and `OM_Environment`[11]. Figure 3.g provides an overview of the standard interfaces.

The consistent functionality across all participating environments is essential but can be problematic for singular interfaces. The singular interfaces `OM_Boolean`, `OM_Integer` and `OM_Float` provide only the minimal functionality that is common among most environments. Additional functionality that is provided by some environments is not supported. The signatures that are defined by these interfaces serve only as substitutes for special syntactical constructs like operators used within most language environments.

[11] A number of additional standard interfaces exist that are not shown here for reasons of space.

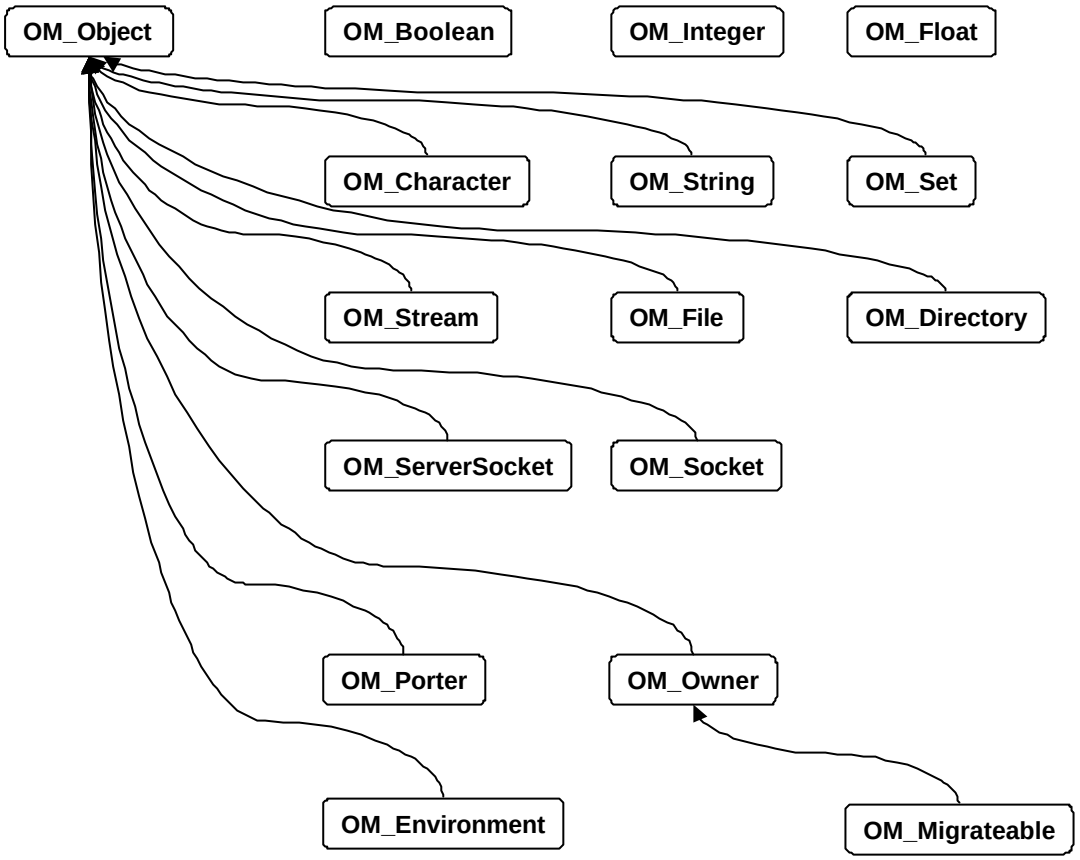

Figure 3.g: The set of standard interfaces used by the HLM migration mechanism. The interfaces `OM_Boolean`, `OM_Integer` and `OM_Float` are singular interfaces while all other interfaces inherit from the generic interface `OM_Object`.

The `OM_Boolean` interface provides only common Boolean operations. The implementation of the interface `OM_Boolean` is usually only affected by syntactic differences of operators and constants that have to be taken care of when a GOAL object definition is translated into a native object definition of a participating environment. Excerpt 3.d shows the interface declaration of the `OM_Boolean` interface.

```
interface OM_Boolean {
     OM_Boolean not();
     OM_Boolean and(OM_Boolean b);
     OM_Boolean or(OM_Boolean b);
     OM_Boolean xor(OM_Boolean b);
};
```

Excerpt 3.d: The interface declaration `OM_Boolean` provides signatures for the most common Boolean operations.

The interfaces `OM_Integer` and `OM_Float` can not be guaranteed to provide equivalent results for arithmetic operations across environments in all cases. Although standards exist that can provide such equivalence, most languages provide their own idiosyncrasies for arithmetic operations. Even with identical interface definitions minimum and maximum values as well as the effects of rounding may differ between platforms. Floating point operations should therefore be used with special caution within Migrateable objects. This problem was also identified for other approaches as stated by Theimer and Hayes:

> “Many programming languages do not specify the semantics of their behavior exactly or in a machine-independent manner. For example, the same floating point operation may yield different results on different machines. Machine dependent values like the size of data types may be available as part of the language, allowing programs to exhibit machine-dependent behavior.” [ThH1991]

A consistent behavior of the `OM_Integer` and `OM_Float` interfaces will only be possible if widespread implementations of common standards can be used. This can not be forced or guaranteed due to the objective that the participating environments should not be changed. Excerpt 3.e shows the `OM_Integer` and `OM_Float` interface declaration.

```
interface OM_Integer {
     OM_Boolean is_equal(OM_Integer i);
     OM_Boolean is_less(OM_Integer i);
     OM_Boolean is_less_equal(OM_Integer i);
     OM_Boolean is_greater(OM_Integer i);
     OM_Boolean is_greater_equal(OM_Integer i);
     OM_Integer plus(OM_Integer i);
     OM_Integer minus(OM_Integer i);
     OM_Integer times(OM_Integer i);
     OM_Float divide_by(OM_Integer i);
     OM_Float toFloat();
};
interface OM_Float {
     OM_Boolean is_equal(OM_Float f);
     OM_Boolean is_less(OM_Float f);
     OM_Boolean is_less_equal(OM_Float f);
     OM_Boolean is_greater(OM_Float f);
     OM_Boolean is_greater_equal(OM_Float f);
     OM_Float plus(OM_Float f);
     OM_Float minus(OM_Float f);
     OM_Float times(OM_Float f);
     OM_Float divide_by(OM_Float f);
     OM_Integer toInteger();
};
```

Excerpt 3.e: The interface declarations for `OM_Integer` and `OM_Float` provide mathematical operations as well as conversion methods.

The signatures defined by the singular interfaces `OM_Boolean`, `OM_Integer` and `OM_Float` are never implemented as such within language environments. Their only role is to define the common functionality that has to be matched by the translation of the GOAL-Representation into a language environment that implements the HLM migration mechanism.

The interface `OM_Object` serves as the root interface of the single inheritance tree that comprises all interfaces that can be used for the definition of Migrateable objects within applications of the HLM migration mechanisms. The interface `OM_Object` provides only three very simple signatures that can be implemented quite differently by the participating language environments. Excerpt 3.f shows the `OM_Object` interface declaration.

```
interface OM_Object {
     OM_Boolean identical(OM_Object o);
     OM_Boolean is_migrateable();
     OM_String interface_name();
};
```

Excerpt 3.f: The interface declaration for `OM_Object` provides only three signatures, i.e. a test for identity, a test whether the object is a migrateable object and a retrieval operation for the name of the interface of the object.

Some of the signatures defined by the interface `OM_Object` have a reflective character, namely `is_migrateable()`and `interface_name()`. These two pieces of information about an object are needed by the HLM migration mechanism during the processing of a migration request. These signatures can be very easily implemented, for example by providing constants for each interface involved.

The alternative of requiring a reflective functionality for each participating language environment appears to be to restrictive as few existing language environments provide reflection (see also

chapter 4 page 206). The aforementioned signatures can nevertheless be implemented using reflective techniques within particular language environments.

Some environments implement characters and strings via language primitives but the interfaces `OM_Character` and `OM_String` are not included in the set of singular interfaces. Due to the differences that can be found among languages in this regard and because of the additional functionality required for these interfaces, the use or wrapper code will be required for their implementation. The signature `set_this()` and `get_this()` are provided in order to initialize objects from atomic values and to generate atomic values from objects respectively.

The interface `OM_Character` inherits from `OM_Object` and provides some basic functionality for handling characters as well as some signatures for parsing purposes. The interface `OM_Character` will have to be implemented as using wrapper code and requires the use of the American Standard Code for Information Interchange (ASCII). Since most language environments use that standard or the superset Unicode this restriction does not seem to be significant. Excerpt 3.g shows the `OM_Character` interface declaration.

```
interface OM_Character : OM_Object {
     OM_Boolean set_this(OM_Character om_c);
     OM_Character get_this();
     OM_Character set_eof();
     OM_Boolean is_equal(OM_Character om_c);
     OM_Boolean is_less(OM_Character om_c);
     OM_Boolean is_less_equal(OM_Character om_c);
     OM_Boolean is_greater(OM_Character om_c);
     OM_Boolean is_greater_equal(OM_Character om_c);
     OM_Boolean is_printable();
     OM_Boolean is_nonprintable();
     OM_Boolean is_whitespace();
     OM_Boolean is_mumber();
     OM_Boolean is_alpha();
     OM_Boolean is_alpha_num();
     OM_Boolean is_id_char();
     OM_Boolean is_eof();
     OM_Boolean is_exponent();
     OM_Boolean eq_tab();
     OM_Boolean eq_lf();
     OM_Boolean eq_cr();
     OM_Boolean eq_blank();
     OM_Boolean eq_not();
     OM_Boolean eq_quotes();
     OM_Boolean eq_data();
     OM_Boolean eq_dollar();
     OM_Boolean eq_percent();
     OM_Boolean eq_and();
     OM_Boolean eq_quote();
     OM_Boolean eq_lb();
     OM_Boolean eq_rb();
     OM_Boolean eq_times();
     OM_Boolean eq_plus();
     OM_Boolean eq_comma();
     OM_Boolean eq_minus();
     OM_Boolean eq_point();
     OM_Boolean eq_slash();
     OM_Boolean eq_colon();
     OM_Boolean eq_semicolon();
     OM_Boolean eq_lt();
     OM_Boolean eq_equal();
     OM_Boolean eq_gt();
     OM_Boolean eq_questionmark();
     OM_Boolean eq_at();
     OM_Boolean eq_lsb();
     OM_Boolean eq_backslash();
```

```
        OM_Boolean eq_rsb();
        OM_Boolean eq_caret();
        OM_Boolean eq_underscore();
        OM_Boolean eq_backquote();
        OM_Boolean eq_lrb();
        OM_Boolean eq_or();
        OM_Boolean eq_rrb();
        OM_Boolean eq_tilt();
        OM_Integer asInteger();
        OM_Boolean fromInteger(OM_Integer i);
};
```

Excerpt 3.g: The interface declaration for `OM_Character` offers basic construction, conversion and comparison functionality in addition to the signatures inherited from the `OM_Object` interface.

The interface `OM_String` manages sequences of `OM_Character` objects. Because it is intended to offer on the fly changes in string length it should be implemented through a suitable data structure that offers dynamic allocation and deallocation within each participating language environment. Some of the more unusual comparison functionality offered by `OM_String` is also used for parsing purposes. Excerpt 3.h shows the `OM_String` interface declaration.

```
interface OM_String : OM_Object {
        OM_Booleam set_this(OM_String om_s);
        OM_String get_this();
        OM_Boolean is_empty();
        OM_Boolean is_equal(OM_String om_s);
        OM_Boolean is_less(OM_Character om_c);
        OM_Boolean is_less_equal(OM_Character om_c);
        OM_Boolean is_greater(OM_Character om_c);
        OM_Boolean is_greater_equal(OM_Character om_c);
        OM_Boolean contains(OM_Character om_c);
        OM_Boolean contains(OM_String om_s);
        OM_Boolean starts_with(OM_String om_s);
        OM_Boolean ends_with(OM_String om_s);
        OM_Boolean start_of(OM_String om_s, OM_Integer om_i);
        OM_Integer string_length();
        OM_Character at(OM_Integer i);
        OM_Boolean string_append(OM_Character om_c);
        OM_Boolean string_append(OM_String om_s);
        OM_Boolean string_append(OM_Integer om_i);
        OM_Boolean string_append(OM_Float om_f);
        OM_Boolean string_append_newline();
        OM_Boolean string_append_tabs(OM_Integer om_i);
        OM_String string_copy();
        OM_String nth_substring(OM_Integer om_i);
        OM_String string_left(OM_String om_s);
        OM_String string_right(OM_String om_s);
        OM_Character asCharacter();
        OM_Boolean fromCharacter(OM_Character c);
        OM_Integer asInteger();
        OM_Boolean fromInteger(OM_Integer i);
        OM_Float asFloat();
        OM_Boolean fromFloat(OM_Float f);
        OM_Boolean print_to(OM_Stream s);
};
```

Excerpt 3.h: The interface declaration for `OM_String` builds on the `OM_Character` interface and offers comparison, conversion and output functionality.

The `OM_String` interface includes an output operation in form of the `print_to (OM_Stream s)` signature. Conversion functionality for the more fundamental interfaces is also provided. The interface `OM_String` plays a pivotal role in the generation or the various representations that

are used by the HLM migration mechanism. The conversion functionality for the singular objects is located differently within various languages. Therefore it has to be integrated in the wrapper code of the `OM_String` interface by the implementation of the HLM migration mechanism for a particular language environment.

The other standard interfaces will only be briefly mentioned here as their declarations and in depth discussion takes up to much space but would add little value. The remaining standard interfaces not already described are: `OM_Set`, `OM_Stream`, `OM_File`, `OM_Directory`, `OM_ServerSocket` and `OM_Socket`.

The `OM_Set` interface defines the usual mathematical operations like union, intersection and difference for sets of objects of any interface derived from `OM_Object`. Signatures for iterations through the whole set of elements are offered as well as per-element operations for union and difference including a contains test. Additional interfaces that inherit from `OM_Set` are defined in the context of th emigration architecture, namely `OM_OwnerSet` and `OM_MigrateableSet`.

Limited input/output operations based on the `OM_Character` and `OM_String` interfaces are provided by the `OM_Stream` interface and similar operations are offered by the `OM_File` interface for the management of background storage. The interface `OM_Directory` provides a simple management of file system structures.

Network operations based on the TCP/IP standard are supported by the `OM_ServerSocket` and `OM_Socket` interfaces. The `OM_ServerSocket` can be used to listen for incoming communication requests while the `OM_Socket` interface can be used for actual communications on either server or client side. A complete listing of the interface declarations of the standard interfaces is provide in the appendix (see page 154).

The objects of the migration architecture of the HLM migration mechanism, are also defined as standard interfaces as the mechanism itself is operating in the same way across all participating environments. The platform-independent nature of the migration mechanism becomes apparent through the fact that the HLM migration mechanism is solely implemented in GOAL except for environment specific parts. The interfaces of the migration architecture are named `OM_Porter`, `OM_Owner`, `OM_Migrateable` and `OM_Environment`.

The `OM_Environment` interface is implemented natively as it encapsulates the platform-specific parts of the migration mechanism like the initiation of the target code generation, the compilation of the target code, the dynamic binding of the resulting object code and the dynamic creation of objects The details of the interfaces of the HLM architecture will be described throughout the following sub chapter that explains the migration mechanism in detail.

3.5 Algorithm

The heart of the HLM migration mechanism is the *migration algorithm* that actually performs the migration. The migration algorithm operates on two levels, the environment level and the object level. The sequence of messages that are exchanged among the participating environments throughout the migration process is call the *migration protocol*.

Within both the source and destination environment the objects of the migration architecture employ synchronous message passing to prepare the information that needs to be transferred between the systems as well as to process the transferred information. Between the participating environments messages are exchanged using different communication means in order to actually transfer the necessary information and to control the whole migration process.

The migration algorithm of the HLM migration mechanism is implemented as part of the behavior of the `OM_Porter` interface and consists of the following six phases that are executed in cooperation of the `OM_Porter` objects of the particular source and destination environments. The migration algorithm is invoked upon the receipt of a migration request by the `OM_Porter` object of the source environment.

1. **Initiation**

   The *initiation* phase ensures the communication between the source environment and the destination environments specified by the migration request.

2. **Check**

   The *check* phase tests whether the object to be migrated as well as its related objects agree with the migration and determines the set of related objects that need to be migrated as well.

3. **Negotiation**

   The *negotiation* phase determines the requirements of the object to be migrated and its related objects and identifies the semantics that need to be added to the destination environment.

4. **Transfer of Semantics**

   The *transfer of semantics* phase makes the necessary semantics available within the destination environment.

5. **Transfer of State**

   The *transfer of state* phase recreates the migrated objects within the destination environment.

6. **Completion**

   The *completion* phase commits the migration.

The migration algorithm of the HLM migration mechanism will abort the migration if an error occurs during any phase and the subsequent phases will then not be executed. These six phases of the migration algorithm are described in detail in the following subchapters.

3.5.1 Initiation

In the context of the HLM migration mechanism, the migration of an `OM_Migrateable` object to another environment is initiated through a message, called *migration request*, to the `OM_Porter` object of the source environment. The migration request needs to specify the object to be migrated as well as a destination for the migration.

In the context of location independent object references an `OM_Owner` object of the destination environment could be specified directly as the target of the migration. Since the prototypical implementation of the HLM migration mechanism does not support location transparency the simplest form of location information available is used.

A migration request has three parameters: a local reference to the object to be migrated, called the *root object* of the migration request, an network address and the port number of the popular TCP/IP network protocol at which the `OM_Porter` object of the destination environment is supposed to be listening. This scheme can also be applied to migration between environments running on a single host. Excerpt 3.i shows a code fragment containing a migration request.

```
OM_Porter ap;
OM_Migrateable am;
...
ap.migrate_via_network(am, "127.0.0.1", 9876);
```

Excerpt 3.i: An actual migration of the HLM migration mechanism is initiated through a migration request. In the above example a message is sent to the `OM_Porter` object `ap` including a reference to the `OM_Migrateable` object `am` and the Internet address and port number of the `OM_Porter` object of the destination environment.

A migration request can be the result of an end-user interaction with an on-screen representation of an object or may be caused by a commando given to a command-line interface. Also, an application may request a migration due to some criteria of the application

logic. The only alternative that is ruled out, is a migration request that is sent by the OM_Migrateable object itself to the OM_Porter object.

A migration request by an OM_Migrateable object is not feasible in the context of the HLM migration mechanism as the OM_Migrateable object is supposed to be inactive, i.e. it can not be referenced by activation records. Consequently, an initiation of migration through a message to the object to be migrated like in many migration systems is not possible in the context of the HLM migration mechanism.

Even if the object to be migrated would not control the migration process itself but rather delegate the control to the OM_Porter object, for example through a message, the object would not be inactive as it would still be referenced by an activation record. The necessary provisions to make sure that the object to be migrated is not referenced by activation records are discussed in the following subchapter (see page 96).

An alternative for the initiation of a migration via the OM_Porter object would be a message to an OM_Owner object of the object to be migrated. As there is no distinct OM_Owner object in most cases an arbitrary one would have to be chosen. Even in cases where there is only one OM_Owner object of a OM_Migrateable object, that OM_Owner object would still have to delegate the control of the migration process to the OM_Porter object. As the OM_Porter object is designed to be the sole entity that controls the migration process within the migration architecture it has to be addressed directly by a migration requests.

In response to the migration request the OM_Porter object checks whether another migration requests is already being processed and aborts the newly received request if that is the case. The processing of every migration request has to be committed or aborted before another request can be serviced. An additional migration request may for example be erroneously invoked during the processing of a prior migration request by one of the participating objects.

The initiation phase continues with an initial exchange of message between the two OM_Porter objects in order to verify communication. The OM_Porter object of the source environment checks whether a OM_Porter object is actually available with the given destination environment and whether it is capable of performing the migration, i.e. not busy with another migration. If the OM_Porter object of the destination environment can not be reached after a timeout period the current migration is aborted.

If the migration request can be serviced and a communication has been established a so called *migration identifier* is generated and used to identify the current migration. The OM_Porter object of the source environment then continues the migration with a check of the object to be migrated as the next phase of the migration algorithm.

3.5.2 Checks

Once a migration has been initiated a *migration check* is performed to ensure that the OM_Migrateable object specified by the migration request and that all related objects either approve the migration or can be migrated as well. As a result of the check the so called *migration set* that consists of all related objects that have to be migrated together with the root object of the migration request is constructed. Prior to the migration check a so called *inactive check* needs to be performed that is described later.

Migration Check

The decision whether an OM_Migrateable object is ready to be migrated and whether an OM_Owner object is ready to let a related OM_Migrateable object migrate is made by the respective objects themselves based upon their current state and through application dependent messages that are sent between the respective objects.

The OM_Porter object sends a ready_to_migrate() message to the root object of the migration request including the migration identifier and an OM_MigrateableSet object that represents the migration set as parameters. The migration identifier is used to distinguish the

current migration request from previously aborted migrations. The `OM_Migrateable` object checks its local state and sends appropriate messages to its related objects.

The `OM_Migrateable` object will send `ready_to_let_component_migrate()`messages to all its owners to make sure that all owners agree with its migration. The `OM_Migrateable` object will return false to the `OM_Porter` if a single owner disagrees and the `OM_Porter` will abort the migration.

If all its owners agree the `OM_Migrateable` object will send either `ready_to_migrate()` or `ready_to_let_owner_migrate()` message to its non-singular components depending on whether it requires a particular component to be migrated as well or not. If a component that is sent a `ready_to_let_owner_migrate()` message disagrees, the `OM_Migrateable` object can send the component an additional `ready_to_migrate()` message to test whether the component will accept to be migrated itself.

If a component that is sent a `ready_to_migrate()` message disagrees the `OM_Migrateable` object has to return false to the `OM_Porter` which will abort the migration. Excerpt 3.j shows a fragment of the object definition of a `OM_Migrateable` object that illustrates the implementation of the migration check.

```
interface My_Migrateable : OM_Migrateable {
...
OM_OwnerSet owners;
OM_Migrateable component1;
OM_Migrateable component2;
...
OM_Integer current_om_id;
...
OM_Boolean ready_to_migrate(OM_Integer om_id, OM_MigrateableSet ms){
     ...
     if (current_om_id == om_id) {
          return true;
     } else {
          current_om_id = om_id;
     };
     // local state check ...
     if (!owners.ready_to_let_component_migrate(om_id, this)) {
          return false;};
     if (!component1.ready_to_migrate(om_id, this, ms)) {
          return false;};
     ...
     };
     if (!component2.ready_to_let_owner_migrate(om_id, this)) {
          if (!component2.ready_to_migrate(om_id, this, ms)) {
               return false; };
     };
     ...
     ms.union(this);
     return(true);
};
...
};
```

Excerpt 3.j: In the above example the object definition of an interface called `My_Migrateable` extends the `OM_Migrateable` interface that defines the `ready_to_migrate()` signature. The signature `ready_to_let_component_migrate()` of the `OM_OwnerSet` interface is used to check each `OM_Owner` object registered with the `My_Migrateable` object iteratively. If all owner objects agree the components of the `My_Migrateable` object will be checked recursively using their `ready_to_let_owner_migrate()` and `ready_to_migrate()` signatures.

An OM_Migrateable object has a component called owners of interface OM_OwnerSet that is used to register all OM_Owner objects related to the OM_Migrateable. The OM_OwnerSet interface offers a ready_to_let_component_migrate() signature that iterates through the set and sends each element a ready_to_let_component_migrate() message including the migration identifier and a reference to the OM_Migrateable object that initiated the iteration as parameters.

The owners of an OM_Migrateable object are checked first because there are usually only few owners per OM_Migrateable object and the owners usually do not need to ask other objects recursively. As one OM_Owner object may "own" several OM_Migrateable objects an OM_Owner may be asked several times in the context of a single migration, each time for a different OM_Migrateable object.

The designer can apply optimizations internal to an OM_Owner object like caching of the migration identifier, to avoid repeated computations. Nevertheless, the OM_Owner has to be asked several times as it may decide differently for each OM_Migrateable object. A reference to the current OM_Migrateable object being checked is therefore included in the ready_to_let_component_migrate() message sent to the OM_Owner object as well[12].

All these messages include the migration identifier as well as a reference to the OM_Migrateable object that sends the message. The last parameter is necessary as an owner or component of an OM_Migrateable object may reference several OM_Migrateable objects as components or owners respectively and has to be able to determine which has sent the particular message.

The ready_to_let_owner_migrate() messages that are sent to components of an OM_Migrateable object include these parameters as well for the same reasons. The ready_to_migrate() message that are sent to components of an OM_Migrateable object include additionally the reference to the OM_MigrateableSet object as a parameter that has been passed to the OM_Migrateable object in the first place (the reason will be explained in the next section below).

The check whether an OM_Migrateable object is ready to be migrated is implemented as a recursive descent from the root object of the migration request. In order to handle cyclical structures the OM_Migrateable object may cache the migration identifier and return true if it has already been traversed for the current migration. The ready_to_let_owner_migrate() signature of the OM_Migrateable interface that is used in the recursion is defined analogous to the ready_to_let_component_migrate() signature described above.

Collection of the Migration Set

If all owners and components of a OM_Migrateable object agree to its migration the OM_Migrateable object will add itself to the OM_MigrateableSet object that was passed as a parameter in the ready_to_migrate() message. This object represents the migration set, i.e. the set of all objects that need to be migrated together with the root object of the migration request.

The root object decides which of its components should be included in the migration set or should remain within the source environment by either sending it a ready_to_migrate() message or a ready_to_let_owner_migrate() message respectively. If the component is sent a ready_to_migrate() message and agrees to the migration of the root object, it has already added itself to the migration set and will be migrated as well.

The migration check, the determination which related objects are to be migrated together with the root object of a migration request and the construction of the migration set are all performed with a single recursive descent. The recursive descent can further be optimized as objects that

[12] The OM_Porter uses the same signature for the initial ready_to_migrate() message to the root object of the migration request with a reference to the root object as the parameter. This detail has been omitted from the code fragments in order to avoid confusion.

have already been visited can detect this using the migration identifier and do not necessary have to perform their own checks again unless their semantics requires to do so.

As part of the recursive descent a `OM_Migrateable` object may also receive a `ready_to_let_owner_migrate()` message from one object and a `ready_to_migrate()` message from another object. As the second check has precedence over the first the recursive checks will have to be performed as well. The `OM_Migrateable` object may nevertheless decide differently for each invoked check.

If an `OM_Migrateable` object has already agreed to a `ready_to_migrate()` message it may simply return true if its sent a `ready_to_let_owner_migrate()` message. The code for the construction of the object representations and for the recreation of the objects within the destination environment is able to cope with such a situation (see also page 109).

Inactive Check

Prior to the migration check and the collection of the migration set an additional inactive check need to be performed by the migration algorithm of the HLM migration mechanism. As mentioned previously the `OM_Migrateable` objects that are supposed to be migrated have to be inactive, i.e. they can not be referenced by activation records as these references will be broken when the objects are migrated.

If an applications of the HLM migration mechanism is not designed in such a way that a migration can only be requested when the corresponding `OM_Migrateable` objects are inactive the check phase of the migration algorithm has to be able to detect previous invocations of the root object of the migration request and its related objects.

This is necessary because any invocations of methods of the `OM_Migrateable` object in question will require the object to be available when the subsequent invocations that eventually requested the migration return. Any attempted access to the `OM_Migrateable` object in this regard will fail if the object has been migrated in the meantime.

Likewise any invocations of methods of `OM_Owner` objects of the `OM_Migrateable` object in question that take the `OM_Migrateable` object as a parameter may try to access that parameter and will fail if the object has been migrated in the meantime. Both kinds of invocations will need to be detected prior to migration by what can be called an *invocation check* in order to inhibit the current migration from being performed. Figure 3.h illustrates the stack of activation records including unacceptable activation records at the time of migration.

The migration algorithm of the HLM migration mechanism can perform the detection of previous invocations of `OM_Migrateable` objects in one of the following two ways:

- **Stack Traversal**

  The straightforward way to detect unacceptable invocations that occurred prior to the migration request would be a traversal of the activation records of the execution stack at the time of migration. Unfortunately only very few language environments as for example Beta [MMN1993] and Loops [SB+1983] provide programmatic access to activation records.

- **Invocation Counter**

  The detection of unacceptable activation records can also be achieved through the use of invocation counters that are incremented with any relevant invocation and decremented upon its return. The detection of an unwanted previous invocations is thus reduced to the test whether the relevant invocation counters have values greater then zero.

Due to the first objective that precludes changes to the participating language environments the second alternative is chosen for the HLM migration mechanism. Invocations counter have to be maintained consistently for all `OM_Migrateable` objects as well as all components of `OM_Owner` objects separately.

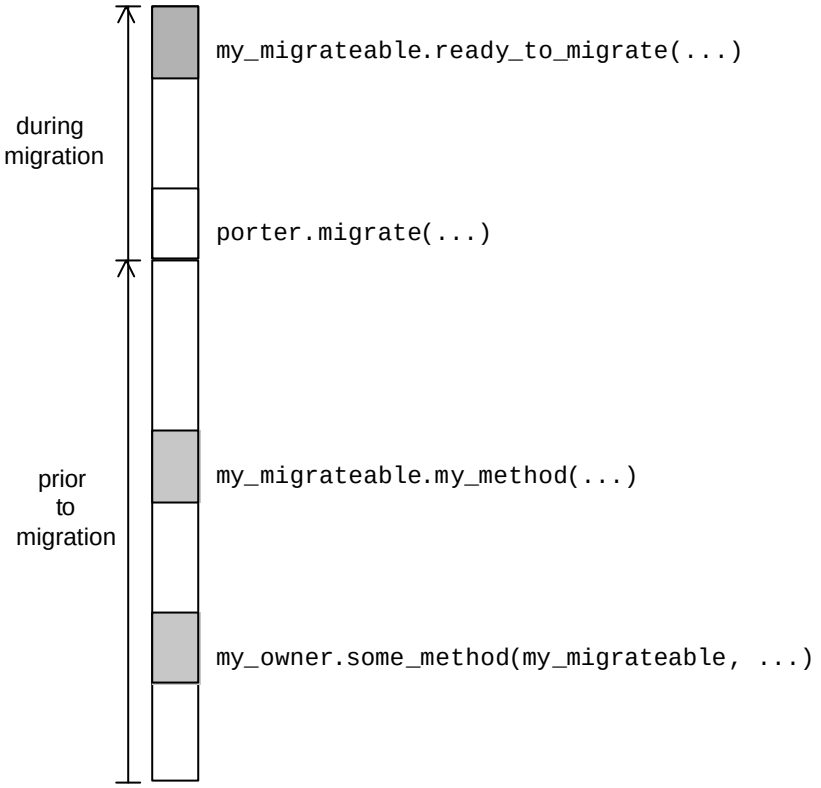

Figure 3.h: The stack of activation records at the time of migration consists of the method invocations that are necessary to process the migration as well as of the invocations that occurred prior to the migration request `porter.migrate(my_migrateable,...)` in the above example. Any invocation of methods of the object to be migrated like `my_migrateable.my_method()` as well as any invocation of an owner of the object to be migrated that uses that object as a parameter like `my_owner.some_method(my_migrateable,...)` are unacceptable.

The designer of an `OM_Migrateable` object has to ensure that a invocation counter is incremented with every invocation and decremented just before a return from any invocation within the method code of every signature of the `OM_Migrateable` object. Excerpt 3.k shows an example of the necessary code for the invocation counter of `OM_Migrateable` objects.

```
interface My_Migrateable : OM_Migrateable {
OM_Integer invocations;
...
OM_Boolean my_method(){
     invocations = invocations + 1;
     if (...) {
          invocations = invocations - 1;
          return true;
     } else {
          invocations = invocations - 1;
          return false;
     };
};
```

Excerpt 3.k: In the above example the interface `My_Migrateable` defines a signature `my_method()` that increases the `invocations` counter of the `My_Migrateable` object upon invocation and decreases it for each possible return.

A similar invocation counter can be used by `OM_Owner` objects for each of their `OM_Migrateable` components. During the execution of an application one of the `OM_Owner`

objects of an OM_Migrateable object may use its reference to the OM_Migrateable object as a parameter in an invocation of a method of another object.

Since the reference is passed outside of the control of the migration architecture the invocation counter for that component will ensure that no migrations of the corresponding OM_Migrateable can be performed until that invocation returns. Any subsequent invocations for that component will be accounted for as well.

An invocation counter must be used not just a simple flag as the methods that increment the counter may be called recursively. A single invocation counter per component of an OM_Owner will be sufficient though. Separate counters per signature or even for each call that passes a reference to an OM_Migrateable are not necessary.

The invocation counter for the OM_Migrateable component of the OM_Owner object will have to be increased prior to each invocation that takes that component as a parameter and decreased when the invocation returns. Excerpt 3.l shows an example of the necessary method code for the invocation counters of OM_Owner objects.

```
interface Some_Interface {
...
OM_Boolean some_method(My_Migrateable my_migrateable,...){
     ...
};

interface My_Owner : OM_Owner {
...
My_Migrateable component1;
Some_Interface some_object;
OM_Integer component1_invocations;
...
OM_Boolean any_method(...){
     ...
     component1_invocations = component1_invocations + 1;
     some_object.some_method(my_migrateable,...);
     component1_invocations = component1_invocations - 1;
     ...
};
```

Excerpt 3.l: In the above example the interface Some_Interface defines a signature some_method() that takes an My_Migrateable object as a parameter. The interface My_Owner defines a signature any_method() that invokes some_method(). The counter component1_invocations has to be increased prior to the invocation and decreased when it returns.

As part of an invocation an OM_Owner object may pass a reference to an OM_Migrateable to another object x. That object x is only allowed to invoke methods of the OM_Migrateable or use the reference as a parameter in another invocation. It is not allowed to store that reference as a component effectively creating an unacceptable implicit relationship. Such a behavior is only acceptable if the object x also inherits from the OM_Owner interface and registers with the OM_Migrateable when it stores the reference as a component (see also page 78).

If a reference to an OM_Migrateable object is just passed along as a parameter between non-owner objects the first invocation will be counted by the original OM_Owner object and discounted with the last return of that chain of invocations. Using references to OM_Migrateable objects as parameters is acceptable as long as the first invocation is counted and no “copies” of that reference remain after that invocation returns.

Using invocation counters seems tedious for the developer but the necessary code can be generated statically during development through appropriate tools (see also page 121). Such tools can also help to ensure that all objects that can hold references to OM_Migrateable objects are designed to inherit from the OM_Owner interface.

The invocation counters effectively duplicate some information of activation records which is inaccessible within most language environments. As mentioned briefly before the invocation check is only necessary if the designer of an application is not able to ensure otherwise that a migration request are only issued for inactive objects.

This is usually the case for applications that use event-based graphical user interfaces especially when user interactions like mouse clicks can lead to migration requests. Applications that migrate objects only under well known conditions can be designed to avoid an invocation check altogether (see also page 125).

If an invocation check is necessary the `ready_to_migrate()` signature of an `OM_Migrateable` object will have to be implemented differently then the first version shown in excerpt 3.l. Since a `OM_Migrateable` object is also an owner of its components the necessary check to be performed analog to the `OM_Owner` interface have to be included as well. Excerpt 3.m is a fragment of the interface definition of an `OM_Migrateable` object that shows the implementation of the invocation check.

```
interface My_Migrateable2 : OM_Migrateable {
...
OM_Integer invocations;
OM_OwnerSet owners;
OM_Migrateable component1;
OM_Integer component1_invocations
OM_Integer current_om_id;
...
OM_Boolean ready_to_migrate(OM_Integer om_id, OM_MigrateableSet ms){
     if (current_om_id == om_id) {
          return true;
     } else {
          current_om_id = om_id;
     };
     if(invocations > 0 ) then {return false};                    //1
     if (component1_invocations > 0) then {return false};         //2
     // local state check ...
     if (!owners.ready_to_let_component_migrate(om_id, this) {
          return false;};
     if (!component1.ready_to_migrate(om_id, ms) {return false;};
     ...
     ms.union(this);
     return(true); };
...
};
```

Excerpt 3.m: In the above example the `ready_to_migrate()` signature of the `My_Migrateable2` interface contains an invocation check in the form of a test of an invocation counter (see `//1`) as well as of a test of an invocation counter for each component (see `//2`).

If the test of the invocation counters maintained by an `OM_Migrateable` object for its own invocation as well as for the invocations of its components is positive the usual migration check as well as the collection of the migration set can be performed. As part of the migration check the owner objects of the `OM_Migrateable` object will perform their part of the invocation check.

An `OM_Owner` object that is sent a `ready_to_let_component_migrate()` message performs its part of the invocation check by testing its invocation counter for the `OM_Migrateable` object, which was included as a parameter of the message and which should be identical to one of the components of the `OM_Owner` object. If the `OM_Migrateable` object can be identifier as a component and the corresponding invocation counter is zero any additional tests of local variables of the `OM_Owner` object are performed and the appropriate result is returned. Excerpt 3.n is a fragment of the `OM_Owner` object definition that shows the implementation of the invocation check.

```
interface My_Owner : OM_Owner {
...
OM_Migrateable component1;
OM_Integer component1_invocations;
...
OM_Integer current_om_id;
...
OM_Boolean ready_to_let_component_migrate(
     OM_Integer om_id, OM_Migrateable m){
     ...
     if ((component1 = m ) &
          (component1_invocations > 0)) {                    //1
          return false;}
     else {...};
     // local check
     ...
     };
...
};
```

Excerpt 3.n: In the above example the `ready_to_let_component_migrate()` signature of the `My_Owner` interface contains a test of an invocation counter for the respective `OM_Migrateable` component (see `//1`).

When the recursive invocation and migration check of the root object of the migration request have been positive and the migration `set` has been collected along the way the migration algorithm can proceed with the negotiation phase. Otherwise the migration will be aborted and the `OM_MigrateableSet` object that represents the migration set is deleted.

3.5.3 Negotiation

Since objects are defined through the encapsulation of state and behavior the migration of objects at the language level has to deal with these two interdependent levels of abstraction. Any approach to the migration of language objects has to consider that the state of objects can not be transferred without knowledge whether the corresponding behavior is available within the destination environment. If the behavior is not available within the destination, the necessary functionality needs to be transferred as well in the form of the corresponding object semantics.

The minimal state to be transferred consists theoretically of only the root object of the migration request while the minimum behavior to be transferred consists of only the interface definition of this object. Because an object to be migrated in most cases relies on related objects to be useful, these objects have to be migrated as well and their behavior also needs to be available.

In the context of the HLM migration mechanism all interfaces that are derived from the `OM_Migrateable` interface define a set of interfaces that can be called the *migrateable interface set*. All objects whose interfaces are members of the migrateable interface set can at least potentially be migrated and define a set of objects that can be called the *migrateable object set*.

The transitive closure of the set of related objects of the root object of a migration request that inherit from the `OM_Migrateable` interface defines a set of objects that can be called the *related set*. The set of objects that need to be transferred for a particular migration is determined recursively during the previous check phase and is called migration set. The migration set is a subset of the related set which is a subset of the migrateable set.

As the objects of the migration set are to be migrated their semantics need to be available within the destination environment. The transitive closure of the interface definitions of the objects of the migration set are dependent upon, defines the set of interfaces that have to be available within the destination environment for the objects of the migration set to be able to work within the destination environment. This set of interfaces can be called the *dependent interface set*.

The dependent interface set is not necessarily a subset of the migrateable interface set as it will at least contain some standard interfaces and may include interfaces derived from the `OM_Owner` interface. If different rules are enforced for the design of `OM_Migrateable` objects the dependent interface set may include other interfaces as well (see chapter 4 page 156). Figure 3.i illustrates the relationship between the migrateable and related objects and interfaces.

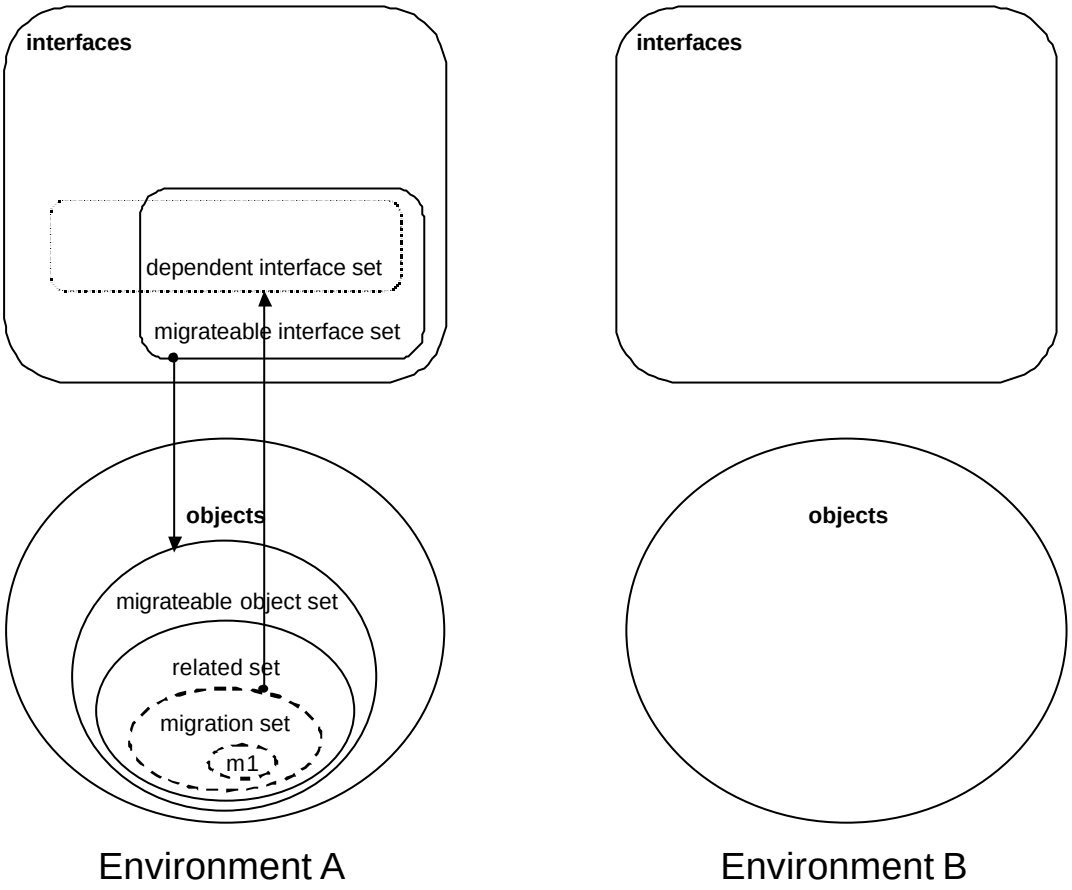

Figure 3.i: The root object of a migration request (object `m1` in the above example) defines the set of objects that need to be transferred during migration recursively, i.e. the migration set. The migration set defines the set of interfaces that need to be available within the destination environment, i.e. the dependent interface set. The migration set is a subset of the set of objects that can be migrated, i.e. the migrateable object set (Environment B is shown here as it will be involved in subsequent phases of the negotiation process).

The minimal migration set consists of the root object of the migration request as the only object, while the maximal migration set may be identical to the migrateable object set. The minimal dependent interface set includes only the interface of the root object of the migration request, while the maximal dependent interface set may comprise all interfaces derived from the `OM_Owner` interface as well as all standard interfaces.

The object migration is possible if the objects of the migration set can be made functional within the destination environment. All interfaces of the dependent interface set therefore need to be available within the destination environment. As not all of these will be available prior to migration, the migration algorithm needs to determine which interfaces are available and whether the other interfaces of the migration interface set can be made available within the destination environment. The HLM migration mechanism uses the extension approach to overcome heterogeneous migration in this regard.

An interface can be made available within the destination environment if all the interfaces it depends upon are available within the destination environment or can be made available within

the destination environment. The respective interfaces are called *extensible* with regard to the destination environment. The negotiation phase of the migration algorithm therefore identifies the interfaces that need to be transferred to the destination environment.

In order to determine which interfaces are available within the destination environment GOAL interface declarations are generated for all interfaces of the destination environment and transferred to the source environment. The resulting set of interface declarations is called *supported interfaces* and is used to determine the set of *extensible interfaces, i.e.* all interfaces of the source environment that can potentially be made available within the destination environment. Figure 3.j illustrates the determination of the set of extensible interfaces.

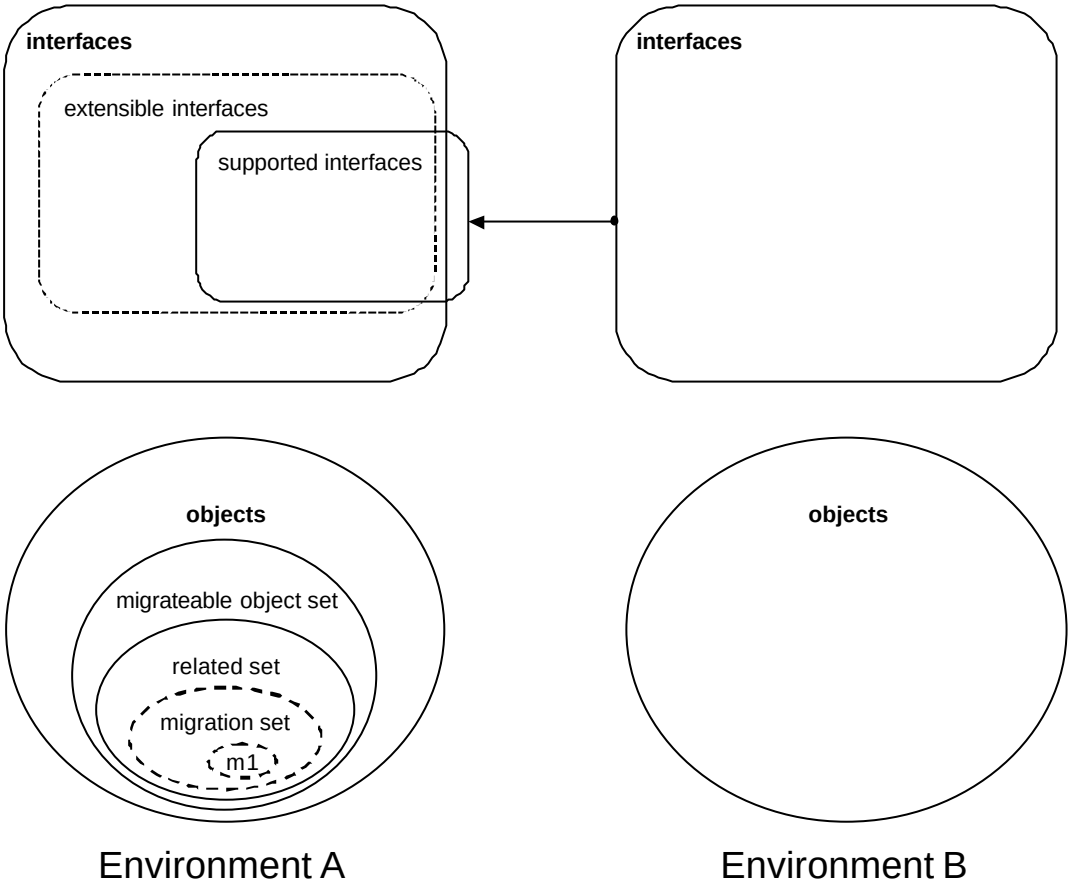

Figure 3.j: The set of extensible interfaces comprises all interfaces that can be made available within the destination environment because their dependent interfaces are contained by the set of supported interfaces, i.e. already available within the destination environment, or can be made available within the destination environment, i.e. are extensible interfaces themselves.

If the dependent interface set is a subset of the union of the set of supported interfaces and the set of extensible interfaces then the migration can proceed. The set of interfaces that actually needs to be transferred is determined as the difference of the dependent interface set without the set of supported interfaces and can be called the *interface set*.

The logic behind the determination of the migration set and the interface set can also be paraphrased as a set of rules for the migrateability of individual objects and interfaces.

An object is migrateable if:

- all related objects it depends upon are migrateable and
- its interface is migrateable

An interface is migrateable if:

- all interfaces it depends upon are supported or migrateable themselves

These rules are applied recursively in order to determine whether an object is migrateable and to construct the set of interfaces to be transferred. Although the interface set is a prerequisite for the transfer of the migration set, there does not need to be a one to one dependence between objects in the migration set and their interfaces being part of the interface set. Figure 3.k illustrates the relationship between the migration set and the interface set for a migration.

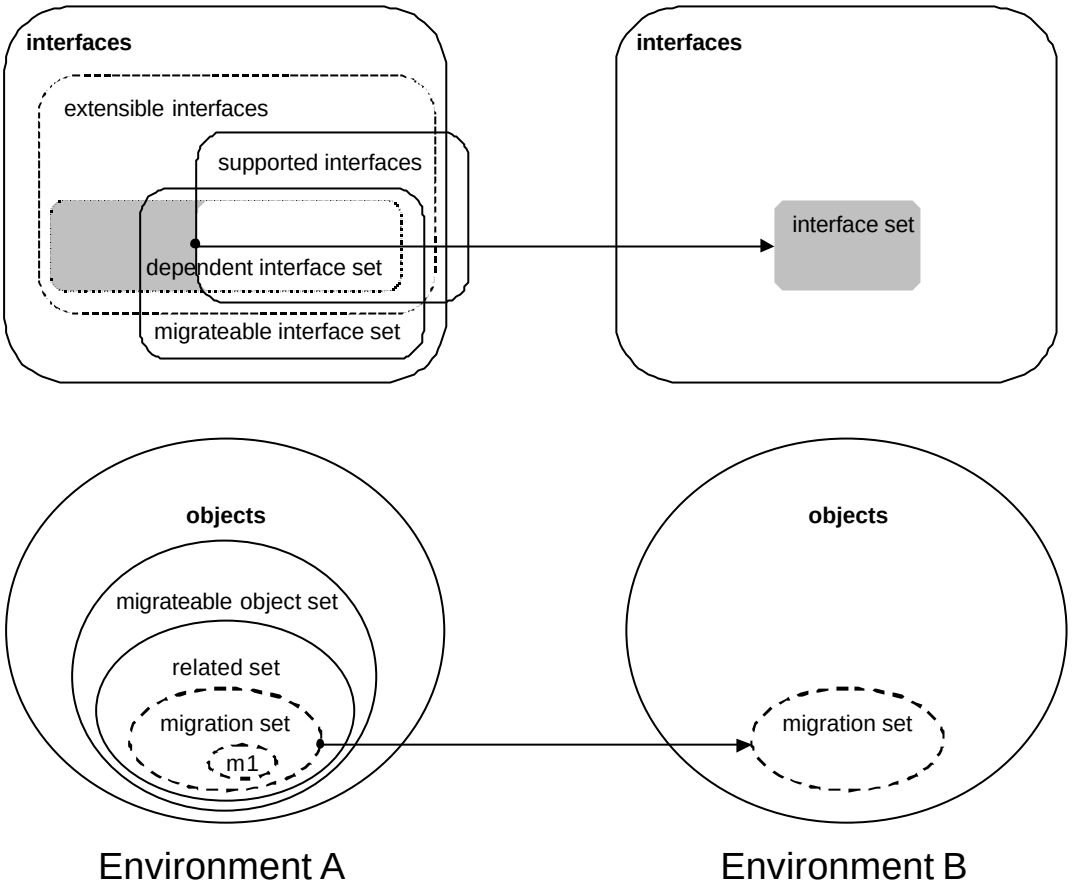

Figure 3.k: The set of interfaces to be transferred to the destination environment called interface set is the intersection of the dependent interface set and the set of extensible interfaces.

The negotiation process just described is the main part of the migration algorithm of the HLM migration mechanism. The prototypical implementation of the negotiation process does not work with the recursive definition of migrateability specified in the above discussion but rather constructs the interface set using a delta iteration in order to achieve a better performance. Excerpt 3.o shows a simplified fragment of the negotiation process.

```
OM_Boolean migrate_via_network(OM_Migrateable m,...){
OM_Integer mid;            //migration identifier
OM_MigrateableSet ms;      // migration set
OM_InterfaceSet si;        // supported interfaces
OM_interfaceSet ip;        // interfaces to process
OM_InterfaceSet is;        // interface set
OM_Interface i;
OM_String in;              // interface name
OM_ComponentSet cs;
OM_Component c;
OM_SignatureSet ss;
OM_Signature s;
OM_ParameterSet ps;
OM_Parameter p;
OM_VariableSet vs;
OM_Variable v;
...
m.ready_to_migrate(mid,m,ms); // check migration and collect migration set
...
si=parse_goal(socket_stream); // receive supported interface set
```

```
...
ip = ms.collect_interfaces();                                          //1
while (i != null) {
      in = i.name();
      if (si.contains(in) == false) {                                  //2
            is.union(i);
            in = i.parent_name();
            if (si.contains(in) == false) {
                  i = this.resolve_interface_name(in);
                  ip.union(i);
            };
            cs = i.components();
            c = cs.element();
            while (c != null ) {
                  in = c.component_interface_name();
                  if ((this.base_interface(in) == false)) &
                        (si.contains(in) == false)) {
                        i = this.resolve_interface_name(in);
                        ip.union(i);
                  };
                  c = cs.next();
            };
            ss = i.signatures();
            s = ss.element();
            while (s != null ) {
                  in = c.result_interface_name();
                  if ((this.base_interface(in) == false) &
                        (si.contains(in) == false)) {
                        i = this.resolve_interface_name(in);
                        ip.union(i);
                  };
                  ps = s.parameters();
                  p = ps.element();
                  while (p != null ) {
                        in = p.parameter_interface_name();
                        if ((this.base_interface(in) == false) &
                              (si.contains(in) == false)) {
                              i = this.resolve_interface_name(in);
                              ip.union(i);
                        };
                        p = ps.next();
                  };
                  vs = s.variables();
                  v = vs.element();
                  while (v != null ) {
                        in = v.parameter_interface_name();
                        if ((this.base_interface(in) == false) &
                              (si.contains(in) == false)) {
                              i = this.resolve_interface_name(in);
                              ip.union(i);
                        };
                        v = vs.next();
                  };
                  s = ss.next();
            };
      };
      i = ip.next();
};
...
```

Excerpt 3.o: A simplified fragment of the `migrate_via_network()` signature of the object definition of the `OM_Porter` interface shows the negotiation process of the HLM migration algorithm. An iteration through the set of dependent interfaces is performed, starting from the interfaces of the objects of the migration set (see `//1`). This iteration determines all interfaces that are not supported (see `//2`) and therefore need to be part of the interface set including the interfaces of ancestors, of components, or signature results, parameters or variables.

The negotiation process is performed by the `OM_Porter` with the interfaces of the objects of migration set as the set of interfaces to process that is called the *negotiation set*. It may seem appropriate to start with the interface of the root object alone, but this would not be sufficient as dynamically created objects would not be handled correctly (see also page 125).

An delta iteration is performed for all interfaces of the negotiation set as follows. If an interface of the negotiation set is not contained in the set of supported interfaces it is added to the interface set. This check is performed iteratively for all components and signatures of the interface. The dependent interfaces that define the components as well as the result, the parameters and the variables of signatures are added to the negotiation set and will be processed by one of the next iterations.

The determination of the interface of an object is based on the appropriate means for each participating environment which will retrieve the name of the interface of an object. The `interface_name()` signature that is declared in the `OM_Object` interface is used for this purpose and need to be implemented appropriately within each participating environment.

The `resolve_interface_name()` signature of the `OM_Porter` object is used to retrieves the interface definition for any given interface name. In the simplest case the interface definitions will be read from GOAL files that have been generated during the development of the application. The prototypical implementation of the HLM migration mechanism manages interface definitions internally as trees of objects that represent syntactical elements.

The negotiation algorithm shown in excerpt 3.o does not check whether the interfaces of components are derived from the `OM_Migrateable` or `OM_Owner` interface. The negotiation algorithm its therefore able to support different design rules that can be developed for future version of the migration mechanism (see also chapter 4 page 156).

The whole negotiation process is controlled through a number of messages that are exchanged between the `OM_Porter` objects of the participating environments. The source environment asks the destination about its supported interfaces. The destination environment generates and sends the GOAL interface declarations to the source and the source environment confirms that it has received the information.

If the GOAL interface declarations could not be received, the set of supported interfaces could not be constructed or the interface set could not be determined, an abort message is sent to the destination environment and the `OM_Porter` of the source will abort the migration. Otherwise the migration process continues with the transfer of the interface set.

3.5.4 Transfer of Semantics

When the negotiation phase has concluded a representation of semantics of the interfaces in the interface set is generated within the source environment using the GOAL language. The `OM_Porter` object conducts an iteration through the interface set and accumulates the representation of the semantics to be migrated in the form of interface definitions.

The generated GOAL source is transferred to the destination environment and the `OM_Porter` object of the destination environment acknowledges the transfer. The representation of the semantics is then parsed and the corresponding source code native to the destination environment is generated and compiled with the help of the `OM_Environment` object of the destination environment.

If the compilation of all interfaces concludes successfully the availability of the interface extensions within the destination environment is reported to the `OM_Porter` object of the source. The migration process then continues with the transfer of a representation of the state of the objects being migrated.

The migration will be aborted though if any of the intermediate steps fail. Occasions for termination include failure to generate the abstract representations, interruptions of the communication, failed parsing of the GOAL interface, inability to generate the native source code of the destination environment or failure to compile the native source code.

If a migration is aborted after the transfer of semantics phase the destination environment is free to decide whether to delete any already successful compiled interfaces or to keep them for possible subsequent migration attempts of the same migration set or for consecutive migrations of different objects.

3.5.5 Transfer of State

Once all necessary interfaces are available within the destination environment, the migration algorithm of the HLM migration mechanism performs the following steps to migrate the objects in the migration set: a representation of the state of the objects is generated, the relationships of the objects of the migration set to other objects of the source environment are deactivated, the representation is transferred to the destination environment, the destination environment recreates the objects of the migration set and reinitializes them.

If the migration is aborted during the transfer of state phase any already recreated objects will be deleted and the relationships of the objects of the migration set to other objects of the source environment are reactivated again. These steps of the transfer of state phase are described in more detail in the following sections.

Representation

A representation of the state of the objects in the migration set is generated within the source environment using the ORL language. In order to transfer the objects between environments, a linearized representation has to be constructed that is able to transfer object graphs of arbitrary complexity including circular structures.

The HLM migration algorithm uses a two phase approach both to generate a linearized representation of the objects in the migration set within the source environment and to reconstruct the migrated objects and the structure of their relationships within the destination environment. Figure 3.p shows a simplified object structure that can be transferred between environments using an ORL representation.

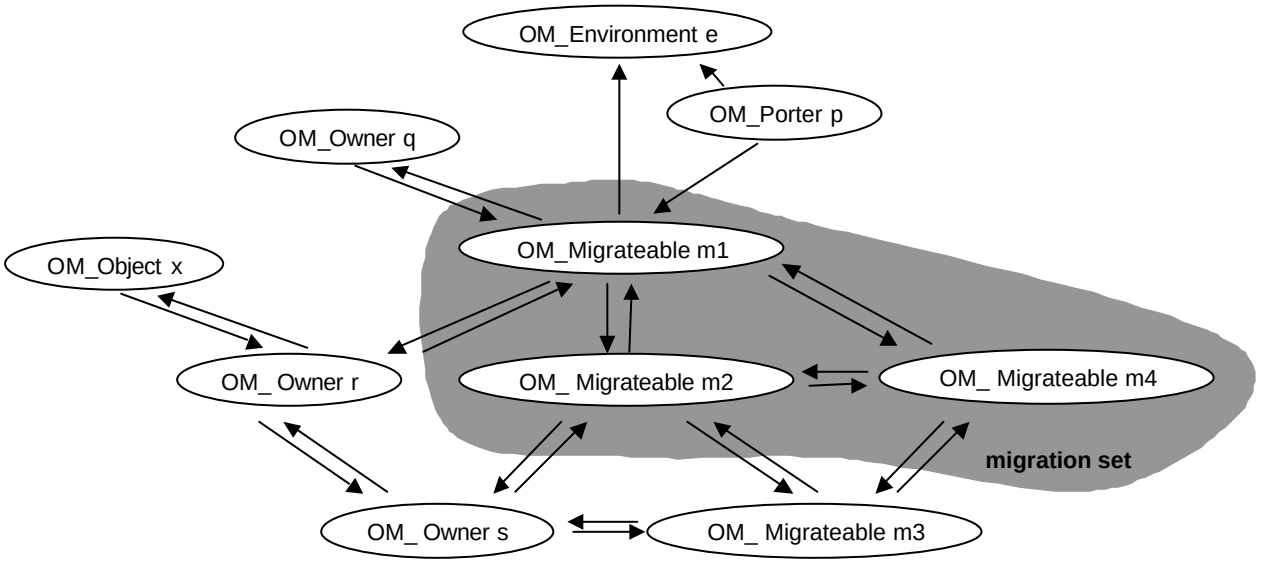

Figure 3.p: An example of a migration set with a cyclical object structure that can be transferred between environments using an ORL representation.

Within the source environment the `OM_Porter` object performs an iteration through the migration set that has been built during the migration check and assigns a unique number called *object migration identifier* to each object in the migration set. For each object the `OM_Porter` queries the interface name and uses it together with the object migration identifier to generate the first part of the ORL representation. This process starts with the root objects of the migration request.

In a second iteration the `OM_Porter` asks the objects in the migration set to generate their own ORL representations. Each object uses its object migration identifier and its components to

generate its individual ORL representation. Each object processes its components in one of the following two ways:

- Singular Objects

  If the component is a singular object, a character based representation of the corresponding atomic value is generated, added to the ORL representation and delimited by a whitespace character.

- Non-Singular Objects

  If the component is a reference to an object of the migration set, the object migration identifier of the referenced object is queried and a colon `:` sign as well as a textual representation of the object migration identifier is added to the ORL representation and delimited by a whitespace character.

The object migration identifiers are used to reestablish the relationships between the migrated objects after their recreation within the destination environment. The use of object migration identifiers to distinguish objects of the migration set within the ORL representation essentially preserves the relative identity of the objects in the migration set during migration.

For each object of the migration set that is processed a lookup for each of its components is performed whether that component is a member of the migration set. This is necessary as an `OM_Migrateable` object may have sent a `ready_to_let_owner_migrate()` message to one of its components during the migration check in order to let it remain within the source environment, but another `OM_Migrateable` object may have sent a `ready_to_migrate()` message to it in order to migrate it as well.

The lookup for membership in the migration set will tell each `OM_Migrateable` object which of its components will actually be migrated. These should be at least all objects that the `OM_Migrateable` itself has intended to be migrated. If this is not the case the `OM_Migrateable` performing the lookup can either cope with the situation gracefully or abort the migration at that point.

Deactivation

After the ORL representation has been generated another iteration through the migration set is performed and a `deactivate()` message is sent to each object of the migration set. This will essentially reduce all mutual relationships of the objects of the migration set to explicit relationships.

In reaction to the `deactivate()` message each object of the migration set will send `deactivate_component()` messages to its owners and `deactivate_owner()` messages to all components that are not members of the migration set. Each of these messages will include a reference to the particular `OM_Migrateable` object that sends the message.

The corresponding `OM_Owner` and `OM_Migrateable` objects will delete their references to the `OM_Migrateable` object passed in the `deactivate_component()` and `deactivate_owner()` message respectively, effectively reducing the mutual relationships to direct ones that are rooted with the objects of the migration set.

This process is called *deactivation.* It effectively deletes all relationships to objects of the migration set from objects outside of the migration set. After deactivation, the source environments is in a ‘ready to commit’ state of a two phase commit protocol that is used to ensure the atomicity of the migration of the whole migration set.

The objects of the migration set themselves still hold references representing the deactivated relationships. This enables them to recover from an abort of the migration during the transfer of state phase. Figure 3.q shows the object structure of the example shown in figure 3.p in the “ready to commit” state, i.e. when the ORL representation has been generated successfully and all objects of the migration set have been deactivated.

Reactivation

The migration process will be aborted during the transfer of state phase if an object of the migration set is not able to generate its ORL representation or the deactivation of an object fails. In the case of an abort the already generated ORL representation will be discarded and the already deactivated relationships will be reactivated.

To accomplish this the `OM_Porter` object will send a `reactivate()` message to all objects of the migration set that have already been deactivated. Those objects will themselves send `reactivate_component()` messages to their owners and `reactivate_owner()` messages to their components that are not members of the migration set. These messages will include a reference to the particular `OM_Migrateable` object of the migration set.

The corresponding `OM_Owner` and `OM_Migrateable` objects will recreate their references to the `OM_Migrateable` object passed in the `reactivate_component()` and `reactivate_owner()` messages respectively, effectively reestablishing the mutual relationships with the objects of the migration set.

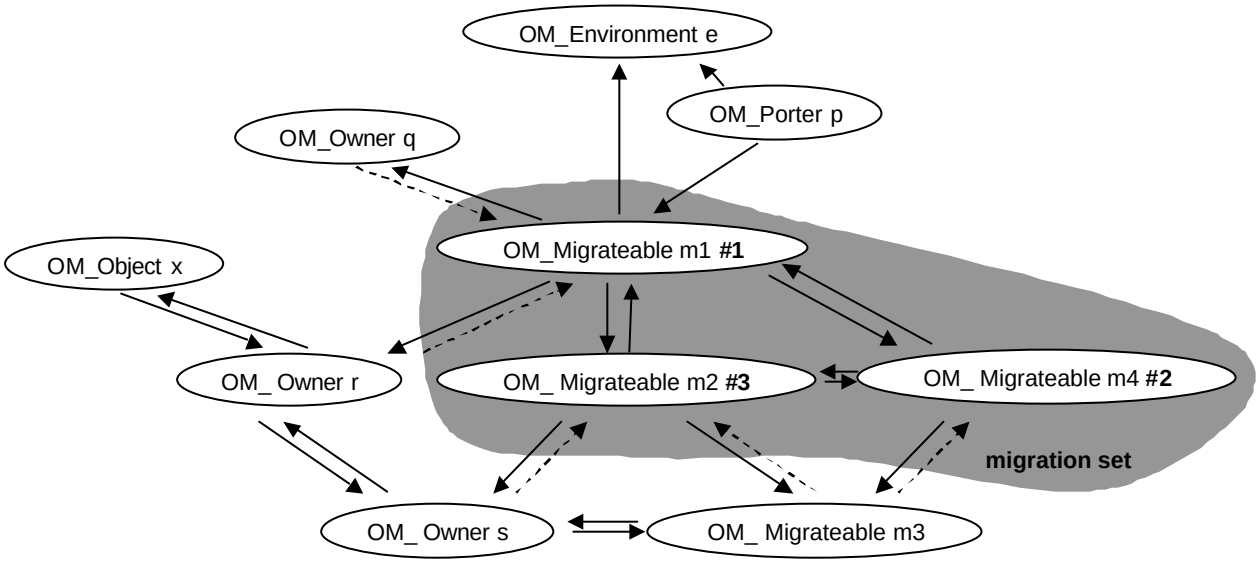

Figure 3.q: An example of a migration set in “ready to commit” state. All objects have been assigned object migration identifiers and after deactivation none of the objects of the migration set will be referenced by any object outside of the migration set (the deactivated references are shown here as dashed arrows). However, the references from the objects of the migration to their related objects outside of the migration set still exist.

If the ORL representation could be generated and all objects of the migration could be deactivated, the ORL representation will be sent to the `OM_Porter` of the destination environment. Excerpt 3.p shows the ORL representation generated for the example in figure 3.p including some illustrative atomic values as components.

```
OM_Migrateable 1
OM_Migrateable 2
OM_Migrateable 3
 |
1 ("test" :3 :2 )
2 (3,1415 :1 * :3 )
3 (9876 :1 :2 * )
```

Excerpt 3.p: The ORL representation of the objects in the migration set consists of the interface names and object migration identifiers as well as of the linearized state of each particular object.

The OM_Porter of the destination environment will acknowledge the transfer of the information. If the transfer of the representation fails for example due to a communication error an abort message will be sent from the OM_Porter of the destination environment to the OM_Porter of the source environment and the objects of the source environment will be reactivated as described above[13].

Rebuild

At the destination environment the first part of the ORL representation is parsed by the OM_Porter object. For each retrieved interface name an appropriate object is created possibly using a previously migrated interface and the corresponding migration object identifier is assigned to the object.

The newly created objects are collected in the so called *rebuild set* of the destination environment. The first object of the ORL representation is identified as the *root object* of the rebuild set that is equivalent to the root object of the migration request within the source environment.

In a following iteration through the rebuild set the OM_Porter object sends a rebuild() message to each object including a reference to the transferred ORL representation. The individual object will read its own ORL representation and restore its components either from the atomic values of the ORL representation or by reestablishing the relationships with other migrated objects. A signature of the OM_Porter provides access to the migrated object corresponding to the migration object identifier.

After this rebuild step the objects of the transferred representation have been recreated within the destination environment and all relationships between these objects have been reestablished using the object migration identifier as the relative identity of the migrated objects in the context of the rebuild set.

Reinitialization

After all objects of the rebuild set have been reconstructed the OM_Porter object sends a initialize_after_migration() message to the root object that includes a reference to the OM_Porter object. This message may also include a reference to the application dependent OM_Owner object designated for the root object within the destination environment. The OM_Porter object will also provide access to the OM_Environment object of the destination environment through appropriate signatures.

The root object of the migration will then perform the operations that have been provided by its designer to ensure that it is working as intended within the destination environment. The root object will send initialize_after_migration() or other appropriate messages to its components recursively in order to ensure that they are working as intended. At this point the migrated objects will only establish unidirectional, i.e. explicit relationships with other objects of the destination environment. Full mutual relationships will be established during the completion phase of the migration process (see next subchapter).

At the end of the transfer of state phase of the HLM migration algorithm the newly established objects at the destination environment will represent the same structure as their still existing counterparts of the source environment. If any parts of the ORL representation can not be read, an object can not be created, an object can not rebuild itself or the reinitialization fails, migration will be aborted as described above.

At the end of the transfer of state phase the OM_Porter of the destination environment sends a message to the source environment indicating that all objects have been rebuilt and that the reinitialization was successful. At this point both environments are in a "ready to commit" state and the completion of the migration can be performed.

[13] The prototypical implementation of the HLM migration mechanism waits indefinites for a network communication to conclude. Alternatively an abort may be issued after a timeout on either side in conformance with the two phase commit protocol.

3.5.6 Completion

The migration algorithm of the HLM migration mechanism performs the completion of a migration if both source and destination environments are in “ready to commit” state. The final "commit" is signaled by the `OM_Porter` object of the source environment to the `OM_Porter` object of the destination environment before the local completion of the migration is performed.

When the `OM_Porter` object of the destination environment receives the “commit” message it sends an `activate()` message including the migration identifier to the root object of the migration. This will establish its intended relationships with for example input/output objects like `OM_Stream` objects via the `OM_Environment` object.

The root object of the migration will also register with its components as an owner and send an `activate()` message including the migration identifier to all of its components which will perform the same operation recursively. Each object of the migration set will use the migration identifier to determine whether it has already been activated, a precaution for cyclic object structures.

At the same time the `OM_Porter` object of the source environment sends a `release()` message to the root object of the migration set within the source environment which will forward this message including the migration identifier to its components recursively. Every object in the migration set will then check via the migration identifier whether it already received a `release()` message. If not, it will delete its atomic values, send its components a `release()` message and delete its references to them.

In the case of a loss of communication after the source has sent the commit message the destination environment will wait in "ready to commit" state until communication is reestablished. The `OM_Porter` object of the source can then resent the commit message and the destination porter will proceed as described while the source `OM_Porter` object may already have released its migration set.[14]

When both environments are in “ready to commit” state an “abort” message may for some reason be sent instead of the “commit” message by the `OM_Porter` object of the source to the `OM_Porter` object of the destination environment. The `OM_Porter` object of the destination environment will then send an `abort_migration()` message to the root object of the migration while the `OM_Porter` object of the source will send a `reactivate()` message to the root object of the migration request.

3.6 Implementation

The prototypical implementation of the HLM migration mechanism is constructed out of several interdependent layers of software. Wherever possible, the HLM migration mechanism is written in the GOAL language itself in order to maximize the portability of the migration mechanism. Only the necessary parts are implemented in the native languages of the participating environments.

The layers that build up the HLM migration mechanism are the standard interfaces, the interfaces that implement the GOAL language processing and the interfaces that implement the migration architecture. Applications that use the HLM migration mechanism define their own interfaces on the basis of the migration architecture, use the standard interfaces and may also use services of the native language environment. Figure 3.r shows the structure of an environment that implements the HLM migration mechanism as well as of an application that uses it.

All standard interfaces are specified via GOAL interface declarations but their implementation differs. The singular interfaces `OM_Boolean`, `OM_Integer` and `OM_Float` will in most language environments not be implemented themselves. Rather they are provided via mappings to existing language constructs as part of the code generation for the particular native language. Alternatively wrappers around existing object definitions can be defined.

---

[14] The resent of the “commit” message is not implemented in the prototypical implementation.

The other standard interfaces including `OM_Character`, `OM_String`, `OM_Set`, `OM_Stream`, `OM_File`, `OM_Directory`, `OM_ServerSocket` and `OM_Socket` can be fully or partially implemented in the native language. The more abstract signatures of these interfaces can as well be implemented independently in the GOAL language and the necessary environment specific code can be generated from the GOAL source.

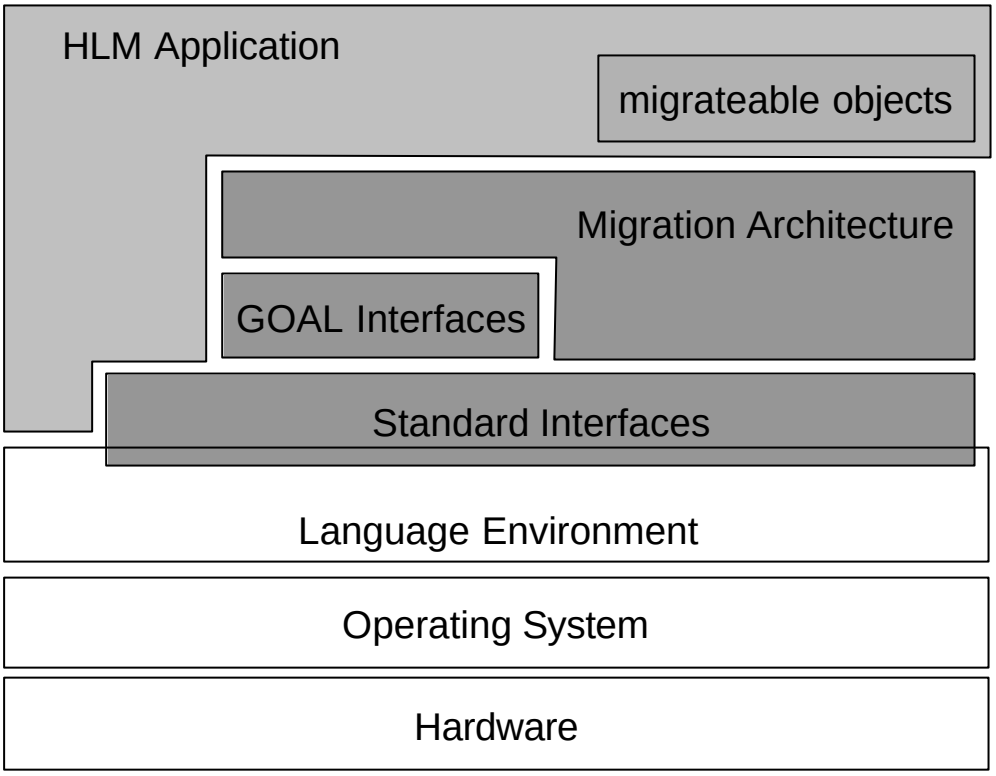

Figure 3.r: The HLM migration mechanism consists of three layers (shown here in gray): the standard interfaces that build the foundation for the main parts of the mechanism, the GOAL interfaces and the migration architecture.

Written solely in the GOAL language are the so called *GOAL Interfaces* that implement the management of GOAL representations based on the basic services of the standard interfaces. The language processing implemented by these interfaces uses an universal scanner to generate a token stream that is the input for a recursive descent parsing mechanism implemented on the basis of one interface for each syntactical construct.

The recursive descent approach is less efficient in the actual processing of the GOAL source code than a comparable stack based parsing mechanism. However, it facilitates the integration of code to generate additional target languages. The interface for each syntactical element of the GOAL language like for example `OM_Interface`, `OM_Statement` or `OM_Expression` only needs to be extended by additional signatures to generate code for the additional language environments (see also the next subchapter page 117).

The GOAL Interfaces are used together with the standard interfaces by the interfaces `OM_Object`, `OM_Porter`, `OM_Owner`, and `OM_Migrateable` that define the migration architecture of the HLM migration mechanism. These interfaces implement the HLM migration mechanism and the `OM_Owner`, and `OM_Migrateable` interfaces can serve as templates for the constructions of applications of the HLM migration mechanism. They are written solely in GOAL source code. Only the supporting interface `OM_Environment` is environment dependent.

The migration algorithm of the HLM migration mechanism is implemented by the `OM_Porter` interface as a series of signatures, one for each phase of the migration process. The decomposition of the migration algorithm into several largely independent building blocks also

helps to support different communication means that can be used to convey the migration protocol between the environments that participate in a migration.

The different phases of the HLM migration algorithm are implemented as separate signatures of the `OM_Porter` interface under the control of the `migrate_via_network()` and the `handle_migration_via_network()` signatures. In the following description of the prototypical implementation of the HLM migration mechanism the word "communicate" is used to indicate that information is exchanged over the network of through any other communication means between the source and the destination environment[15].

1. Initiation

During the initiation phase of a migration the signature `migrate_via_network()` is used by the `OM_Porter` object of the source environment to establish the communication with the `OM_Porter` object of the destination environment and to execute the migration request. The signature `handle_migration_via_network()` is used by the `OM_Porter` of the destination environment to react to the initiation of communication by the source and to perform the destination part of the migration.

2. Check

During the check phase, the signature `migration_check()` is used by the `OM_Porter` object of the source environment to test whether the migration can be performed and to collect the migration set along the way.

3. Negotiation

At the beginning of the negotiation phase a "send supported interfaces" message is communicated to the `OM_Porter` of the destination that uses the signature `supported_interfaces()` to collect the relevant interface declarations. The resulting GOAL representation is communicated to the `OM_Porter` of the source environment and parsed there. The signature `process_migration()` is used by the `OM_Porter` object of the source environment to construct the interface set.

4. Transfer of Semantics

During the transfer of semantics phase the interface definitions of the interface set are collected by the `OM_Porter` object of the source environment and the resulting GOAL representation is communicated to the `OM_Porter` of the destination environment. The transferred interface set is make available by the `OM_Porter` of the destination environment through the signature `implement()`.

5. Transfer of State

During the transfer of state phase the signature `generate_representation()` is used by the `OM_Porter` object of the source environment to construct the representation of the objects in the migration set and the `deactivate()` signature is used to reach the "ready to commit" state. The resulting ORL representation is communicated to the `OM_Porter` of the destination environment which uses the signature `represent()` to rebuild and initialize the migrated objects and to reach the "ready to commit" state.

6. Completion

Both the `OM_Porter` object of the source environment and of the destination environment use the signatures `activate()` and `release()` to make the migrated objects available and to delete the original ones, respectively. The `reactivate()` and `abort()` signatures are used in the case of an abort of the migration at the source to make the original objects available as well as at the destination to perform the necessary housekeeping respectively.

[15] Communications for control and handshaking purposes that occur for each phase have been omitted to avoid confusion.

The `OM_Porter` objects of the two environments that participate in a migration communicate with a number of messages in order to control the migration process. Communication techniques that have been implemented for the HLM migration mechanism include a native network protocol based on the TCP/IP standard as well as the use of common file system services. For testing purposes the migration protocol has also been build using method invocation within a single environment. Figure 3.s illustrates the migration protocol as well as the whole migration process.

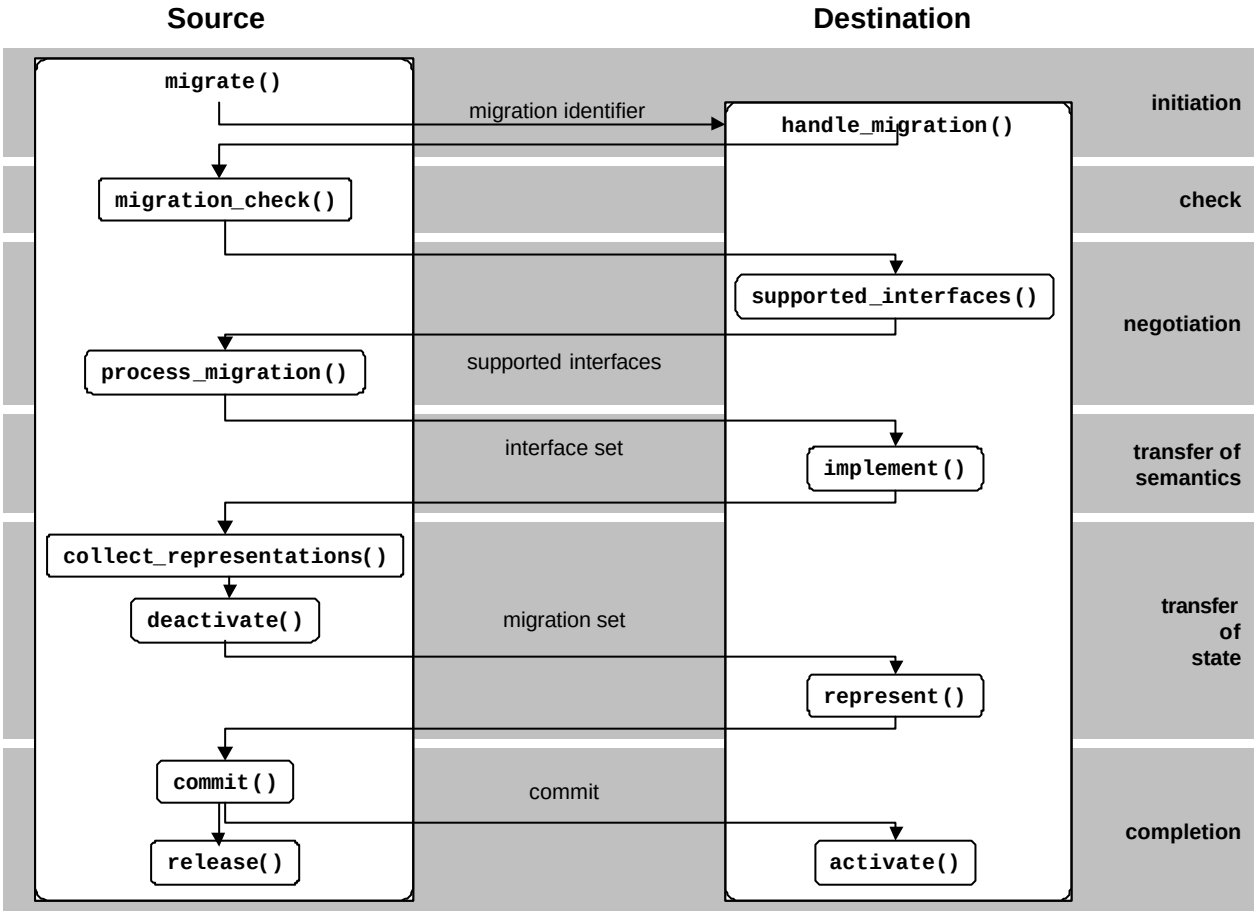

Figure 3.s: The migration process is implemented as a series of method invocations shown here as rounded rectangles alternating with communications shown here as arrows, that are exchanged between environments to control the migration process. Any of the messages communicated between the environments may as well be replaced by an abort message in the case of an error. The phases of the migration algorithm are shown in the background as gray rectangles.

The use of the GOAL language as an implementation language for the HLM migration mechanism was mainly motivated by the increased portability and extensibility. Porting the mechanism to a new language environment is relatively simple and can be done through cross-compilation. The only exception are the standard interfaces that have to be implemented natively (see next subchapter for details).

The migration mechanism as well as the GOAL language can also be extended through the introduction of new standard interfaces (see also chapter four page 154) as well as new negotiation algorithms (see also chapter four page 174) that can become available across all supported platforms rapidly. The layered architecture and the building block approach to the migration process provide the necessary openness to introduce improvements and new ideas.

The downside of the prototypical implementation of the HLM migration mechanism is its lack of efficiency. Since the implementation was intended as a proof of concept rather than as a production system efficiency was not considered high priority. One particular area of

improvement will be the use of stack based parsing techniques as the definition of the GOAL language becomes more mature.

The HLM migration mechanism has been ported to several target languages throughout its design and development using a simple bootstrap approach. The same process can be used to implement the mechanism for additional target languages as described in the following subchapter.

3.6.1 Bootstrapping Object Migration

The prototypical implementation of the HLM migration mechanism was done in the Java Programming Language Environment [GJS1996] and has been subsequently ported to the Common Lisp Object System (CLOS) [Kee1989]. Limited versions of the mechanism have been implemented for the Python programming language [LoF1997].

The following bootstrap process was used to create the Java based prototypical implementation of the HLM migration mechanism. As a first step, a GOAL-to-Java compiler was written in Java and used for the development and the test of a first version of the standard interfaces and the migration mechanism that were defined partially in the Java as well as in the GOAL language.

The Java-based GOAL-compiler was also used as a tool for the reimplementation of the compiler in GOAL, up to the point where the new compiler was able to compile itself. Using the GOAL-based GOAL-to-Java compiler the HLM migration mechanism itself was rewritten in the GOAL language and compiled to Java.

The Java based prototypical implementation allowed the first proof of concept of the HLM migration mechanism. The migration of objects between distributed Java environments that are heterogeneous at the library and application level could be demonstrated. Several problems with an earlier version of the design could be solved.

The HLM migration mechanism was then ported to the Common Lisp Object System (CLOS) through a native implementation of the standard interfaces and a cross-compilation of the migration architecture and the GOAL Interfaces. Although in its third iteration this combined Java/CLOS implementation of the HML migration mechanism is fully functional but still has a prototypical character.

The Java/CLOS implementation of the HLM migration mechanism can be used for a full proof of concept as migration of objects between environments with heterogeneity at the hardware, operating system, language, library and application level can be demonstrated. The HLM migration mechanism has been tested using an interactive application (see also page 125).

Porting

The HLM migration mechanism can easily be ported to additional language environments that fulfill the prerequisites of the mechanism through the following three steps. The new language has to be supported through code generation and the standard interfaces including the `OM_Environment` interface have to be implemented natively. All other elements of the migration architecture can then be cross generated for the new environment.

The first step of the porting process is the extension of the GOAL interfaces with code generating functionality for the new target language. This also simplifies the subsequent porting of the standard interfaces. Because the code generator for the target language is an extension to the existing GOAL interfaces it can be translated into any of the already supported languages.

Using such a cross-compiler, the standard interfaces and the migration architecture can be generated in the source code of the new language within one of the already supported environments. The resulting native source code of the standard interfaces will then have to be extended according to the specifics of the new environment.

Once the porting work for the standard interfaces has been done the additional environment will be able to take part in object migration as a native implementation of the GOAL interfaces and migration architecture can be generated and tested. In order to receive or migrate objects from

or to other environments an application that uses objects that are able to migrate will have to be developed for the new language environment.

3.7 Development using Migration

The HLM migration mechanism imposes a number of constraints on applications which have to be designed specifically for migration. The following subchapter tries to provide some guidelines for the development of new applications that use the HLM migration mechanism as well as for the integration of the migration mechanism into existing applications.

In order to simplify the development of applications a number of design tools are proposed in the second subchapter that can help to satisfy the implementation constraints imposed by the HLM migration mechanism. The integration of support for the HLM migration mechanism into existing development environments is also discussed briefly.

3.7.1 Migration by Design

Applications of the HLM migration mechanism have to be designed specifically for migration due to the chosen objectives. Especially the objectives not to change the objects to be migrated during migration or the participating environments prior to migration impose a number of design constraints for the use of the migration mechanism in practical applications.

Most problems of migration, especially the handling of references between objects, have to be resolved prior to migration within the design of the objects to be migrated as well as within the design of the objects of the surrounding application. As mentioned before the whole approach to migration can be called *migration by design*.

The migration by design approach can be applied within the context of existing environments and to some extend even for existing applications. Alternative approaches to migration that for example implement migration as an operating system feature or as a programming language characteristic orthogonal to type can not be used with existing environments but will require new language implementations or significant changes to existing ones.

The use of the HLM migration mechanism requires that the standard interfaces and the migration architecture are available in the context of the particular language environment and that the corresponding applications are designed for migration. I.e. applications of the HLM migration mechanism either encapsulate some of their functionality as `OM_Migrateable` objects or open themselves to incorporate objects from other environments.

The capability to receive objects that are migrated can be integrated easily into new and to some extend also into existing applications that are redesigned. The migration architecture has to be incorporated into the application and an `OM_Porter` object has to be created that processes migration requests. Depending on the particular application, the `OM_Porter` object may only be created in conjunction with user interactions, for example in reaction to a drop event of a graphical user interface.

Optionally `OM_Owner` objects may be created or some exiting interfaces may be changed or wrapped to conform with the `OM_Owner` interface in order to provide objects the migrated object can interoperate with. Depending on the particular application the migrated objects may well be able to function using only the standard interfaces, for example the input/output operations provided by the common `OM_Environment` object and its related objects.

The development of applications that are able to migrate some of their objects to other environments require more work especially if an attempt to enable migration in existing applications is made. Both the objects to be migrated as well as the objects that will be left behind at the source have to be designed for migration which is sometimes hard to enforce within existing application structures.

According to the migration architecture the objects of an application will have to be either `OM_Migrateable`, `OM_Owner` or unrelated objects. The latter objects are not supposed to participate in the object migration process at all, while the `OM_Owner` objects are used to separate the non-migrateable objects from the migrateable ones.

An application that is created from scratch can be developed with the usual design methodologies but special attention has to be applied to the migration aspect once the basic structure of the application has been identified. Starting with migration as the main purpose of an application will only be reasonable in special cases.

The following guidelines can be used for the design of applications that make use of the migration mechanism. These rules apply to both newly designed applications as well as to existing applications that are extended with migration. The latter case is also mentioned separately within each guideline.

1. Rightsizing of Functionality

The objects that are supposed to be migrateable will have to be chosen with care. These objects should neither be to large nor to small both in terms of their structure, i.e. the number and complexity of their components as well as in terms of their behavior, i.e. the number of methods.

If the objects to be migrated are to large, the probability that all of their functionality will be useful at the destination becomes small and the amount of checking to be done prior to migration will probably not be worthwhile. If objects to be migrated are too small, the overhead necessary to perform their migration might not be worth the limited functionality they add to the destination either.

As a general rule its seems better to have smaller objects that can cooperate rather then to have one large object that is not fully used after migration. If smaller objects are not available, a large one may be broken into smaller ones that can be migrated independently. Each smaller object should implement some significant functionality that it is able to perform even on its own if necessary.

To chose the right objects and the right functionality is especially hard for existing applications that are already confined to a fixed inheritance lattice. In most cases significant changes to the inheritance lattice will be necessary in order to comply with the migration architecture, but these changes might be impossible to implement.

A different approach may be chosen for existing applications if a suitable amount of functionality can be identified within an existing object. The functionality suitable for migration may be factored out into a separate object that is only referenced as a component by the original one. The separate object may then inherit from the `OM_Migrateable` interface to be part of the migration architecture. An `OM_Owner` object will have to be used to connect the two.

2. Minimization of Relationships

Once the objects to be migrateable are identified their relationships to other objects will need special attention. As defined by the migration architecture the related objects need to be either `OM_Migrateable` objects or `OM_Owner` objects that are migration aware. The objects that are related to the `OM_Migrateable` objects have to be chosen with care.

As a general rule the number of relationships of a `OM_Migrateable` object should be confined to the absolute minimum necessary for the object to operate. The migration overhead will be lower the fewer relationships need to be processed for each `OM_Migrateable` object.

References from several objects to an `OM_Migrateable` can be folded into one if an `OM_Owner` object is created as a handle that is stored as a component of the related objects. The objects that use that handle have to be aware of the fact that the relationship to the `OM_Migrateable` object through the handle may be released through migration.

In order to factor out migrateable objects within existing applications, the creation of a pair of an `OM_Migrateable` and an `OM_Owner` object will have the minimal implications for the overall design of the application. An object that is identified as a candidate for migration will therefore be split into three parts: an `OM_Migrateable` object that may actually migrate, an

`OM_Owner` object that obeys the migration protocol and the remaining original object that interacts with the rest of the application as before. The `OM_Owner` object may also be used by other objects as a handle to the `OM_Migrateable` object.

3. Maximization of Interoperation

After the number of relationships of the `OM_Migrateable` objects have been minimized it is reasonable to extend the possible forms of interoperation of the migrateable objects with the potential destination environments. Depending on the kind of application several different approaches more or less independent of the relationships of the source environment can be used.

As a general rule, objects to be migrated should be useful in as many ways as possible as long as their extension in this regard does not add disproportionately to their complexity. As a fallback, a simple interoperation of the migrated objects with a command line user interfaces will be possible using only the standard interfaces.

The inclusion of such a low profile alternative of interoperation will enable migration of objects even into environments that do not provide suitable `OM_Owner` objects to interoperate with. Sophisticated applications will implement specialized `OM_Migrateable` objects that require a specific set of `OM_Owner` objects in order to be functional.

Objects of existing applications that have been identified for migration are more likely to be of only limited usefulness. As no significant changes can be applied to existing objects a predefined set of objects to relate to will be required in most cases even if migrateable functionality can be identified.

Migrateable objects factored out of existing applications will need to be made interoperable using only the standard interfaces in order to be useful within other destination environments. If such an extension is not possible the objects can probably only be migrated to other installations of their original application on different platforms.

The design of new applications can be changed in order to accommodate migration and in some cases applications may be specifically designed for migration. Migration can be an integral part of the design of new applications and will then be available when they are first used. Later extensions to these applications can then also be migration aware.

If migration is added to existing applications the design of some of the interfaces that make up the application will need to be changed. As a consequence existing objects whose data may be stored for example in object database management systems will need to be converted to adhere the changed interfaces.

Some object environments provide automatic tools to convert existing objects to updates of their interfaces. Otherwise the developer who integrates migration into these applications will have to convert the relevant objects. The necessary conversions can be implemented as separate utilities or as an additional initialization phase of the extended application.

If a conversion of existing objects is not possible the integration of migration into existing applications will only yield code reuse but not reuse of the existing objects. If the effort to integrate migration into an existing application will be worthwhile in such a hybrid case depends on the particular circumstances.

Migration of objects may also be possible between new and existing applications in either direction. The probability whether migration can be implemented and performed in each individual case differs largely due to the amount of changes required. Table 3.a provides an overview of all possible cases.

By far the highest probability that migration can be successful exists between new applications that can still be influenced by migration in early stages of their development. The migration of existing objects into new applications can also be implemented comparatively easily as the receiving applications can be extended to a hosting environment.

Significantly harder to implement are migrations from new applications to existing ones. Although newly created objects can be adjusted to some extend to fit into the receiving environment, the existing functionality will probably limit the usefulness of the migrated objects. On the other hand the design of the new application should not be compromised by limitations of other existing software.

	to new applications	**to existing application**
from new applications	high	low
from existing applications	medium	very low

Table 3.a: The probability of migration between new and existing applications depends on the extent of changes that have to be applied. Objects of new application are more likely to be migrateable and new applications are more likely to be able to accommodate migrated objects.

By far the most complicated case is the migration of predefined objects into existing applications which will not be possible at all in many cases. Exceptions will be families of applications that for example share large parts of a common inheritance lattice and are therefore good candidates for successful migrations.

In any case extensive testing of migration with different destination environments will be necessary in order to verify the chosen design. These tests will not only have to include the migration process as such but also the operation of the migrated objects within the destination environments. As with software development in general even extensive testing will not prove whether a migration design will work in a particular case but it will increase the probability.

3.7.2 Development Tools

As a downside of the migration by design approach a number of constraints exist for the development of applications for the HLM migration mechanism. Some requirements have to be considered in the design of the application objects and a significant amount of coding is necessary. However, several tools can be envisioned that simplify the overall design and development process significantly.

Regardless whether new applications are designed or existing applications are extended to take advantage of the HLM migration mechanism, three basic tasks recur in any development of applications of the HLM migration mechanism. New interfaces will have to be created that need to conform with the migration architecture, changed source code fragments will have to be checked for conformance and the changed interfaces have to be tested.

Applications that uses the HLM migration mechanism consist of the application objects that conform to the migration architecture, of some integration code that is necessary to execute the application objects within the particular language environment, and of additionally GOAL interface definitions for the application objects. The additional GOAL interface definitions are a requirement of the prototypical implementation. Future versions of the HLM migration mechanism may be able to generate the necessary interface definitions on the fly or work with different representations formats (see also chapter 4 page 206).

In the context of the prototypical implementation of the HLM migration mechanism, applications that use the HLM migration mechanism can be developed in the following two ways:

- **GOAL based Development**

  The relevant objects of the application can be developed using the GOAL language and the object implementations in the target language are generated from the GOAL source.

- **Target Language based Development**

  The relevant objects of the application can be developed in the target language and the necessary interface definitions in the GOAL language are generated from the source files .

The code that integrates the application objects into the particular language environment has to be developed additionally in both cases. The finished application is running within the environment of the target language while the interface definition can be stored as GOAL files that are accessible by the application if a migration is initiated.

If the application is supposed to be the destination of a migration all relevant GOAL interface definitions supported by the environment need to be available to the application for the negotiation phases of the HLM migration mechanism. The corresponding source code files have to be available as well for the compilation and dynamic load during the transfer of semantics.

The GOAL based development process can be supported by the code generation for the target language that is part of the implementation of the HLM migration mechanism within the particular language environment. The code generator can be also be rebundled as a separate tool that works as compiler from the GOAL language into the particular target language.

GOAL-to-Target Compiler

A *GOAL-to-target compiler* can be used as a tool for the GOAL based development of applications of the HLM migration mechanism. Applications are written in the GOAL language and translate in to the particular target language using this tool. The minimal integration code that is necessary to execute the generated application with the environment of the target language has to be written manually. Figure 3.t illustrates the GOAL oriented development process and the use of a GOAL-to-target compiler.

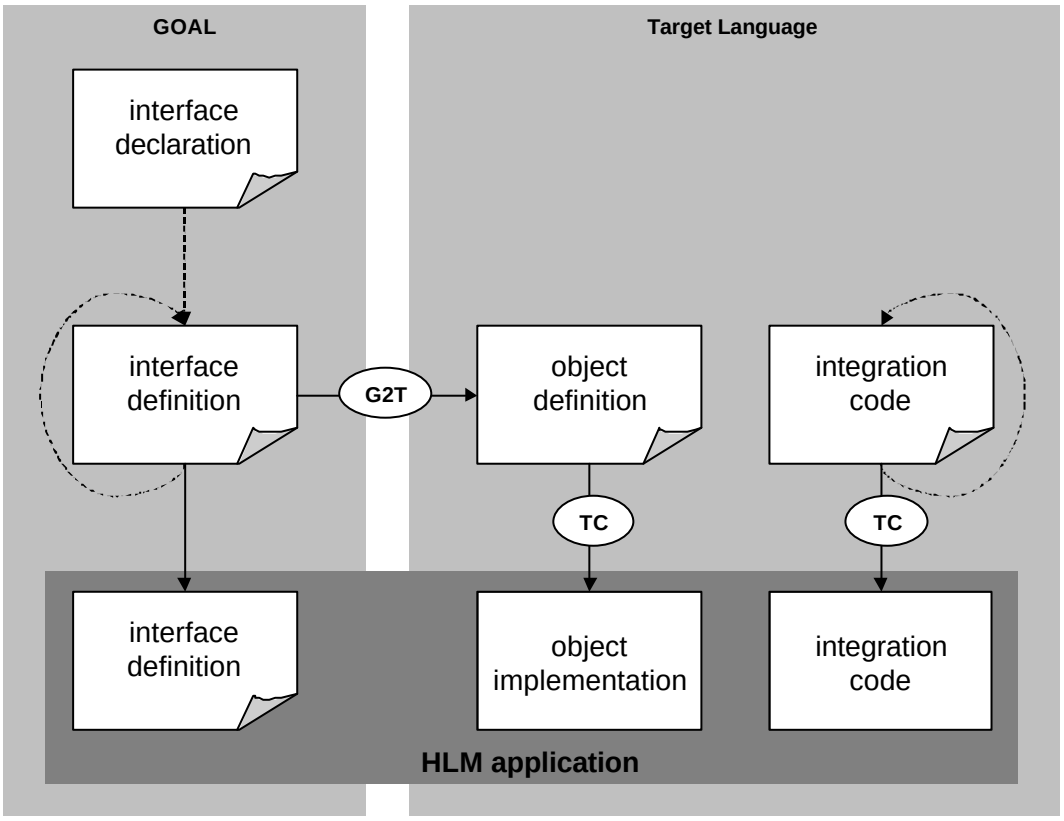

Figure 3.t: The GOAL based development of applications of the HLM migration mechanism uses a GOAL-to-target compiler (G2T) to translate the manually developed GOAL source files into the particular target language (source files are shown here as dog-eared rectangles, manual changes to source files are shown as dashed arrows). The resulting object definitions are compiled together with additional manually developed integration code into an executable with the use of the normal compiler of the target language (TC).

The GOAL based development style was used in the design of the HLM migration mechanism and for its prototypical implementation as well as for the development of some example

applications. Although this development method can be characterized as solid it has some inherent limitations.

The use of the GOAL language as the source language for the development of applications complicates or inhibits the use of functionality that is already available within the particular language environment. Existing functionality has to be integrated manually into the GOAL based applications, which limits the use of the HLM migration mechanism.

Target Language based Development

The target language based development styles does not complicate the integration of existing functionality into applications of the HLM migration mechanism as the particular target language is used as the source language to develop applications. However new tools have to be developed in order to support this alternative development style.

As the application objects that conform with the migration architecture of the HLM migration mechanism are developed in the target language a tool that generates the corresponding interface definition in the GOAL language is necessary. Such a *target-to-GOAL compiler* can also help in the design of HLM compliant applications as it can indicate problems with the target language based implementation.

In order to provide a head-start in the development of application of the HLM migration mechanism in a target language preliminary object definitions of the necessary applications objects could be generated from GOAL object declarations through a tool that can be called *template generator*.

Template Generator

New interfaces that can be defined for applications in the context of the HLM migration mechanism will either have to inherit from the `OM_Migrateable` or the `OM_Owner` interfaces regardless whether directly or indirectly. The basic functionality required by the migration mechanism can be generated in the form of so called *templates, i.e.* new object definitions that have to be filled in with actual application functionality by the developer.

The generated object definitions will have to include all signatures that are necessary for the respective objects to fulfill their role within the migration architecture. Appropriate method code can be generated if additional components and methods are specified in the GOAL interface declarations that are used as input for the template generator. Alternatively the template generator can emit the required additional method definitions for existing object definition of the particular target language[16].

The following methods have to be generated as part of a template:

```
ready_to_migrate()
ready_to_let_component_migrate()
ready_to_let_owner_migrate()
represent()
deactivate()
rebuild()
initialize_after_migration()
activate()
reactivate()
release()
```

The recursive descent implemented by some of these signatures can be generated for all specified components, significantly reducing the effort for the developer. Only minor additions by the developer will be required who will be able to focus on the application specific functionality instead. Optionally, the necessary code to manage invocations counters can also be generated for all relevant signatures by the template generator.

[16] This alternative is not shown in figure 3.u in order to avoid confusion.

The generation of templates is only possible for an initial specification of new interfaces. If additional components or signatures are added to the interface declarations, the resulting object definitions will have to be generated anew. After the initial object definitions have been generated in the target language the interface declarations used as input can be discarded and subsequent manual changes to the object definitions in the target language can be processed by a target-to-GOAL compiler.

Target-to-GOAL Compiler

Apart from the generation of GOAL interface definitions for object definitions of the particular target language, a target-to-GOAL compiler will also be useful to ensure that the source code created by the developer will conform with the migration architecture. Unlike the template generator, the target-to-GOAL compiler will not produce additional code but rather emit error-messages if some of the various tests are not passed.

Due to the complexity of the code that can be added by the developer automatic additions to the code will not be possible but static checks of the given source can be performed. The necessary tests will make sure that the relationships between the objects involved are handled correctly, for example via mutual relationships to `OM_Owner` interfaces and that all components are taken care of in the recursive migration check. Figure 3.u illustrates the use of the template generator and the target-to-GOAL compiler in the context of the target language based development style.

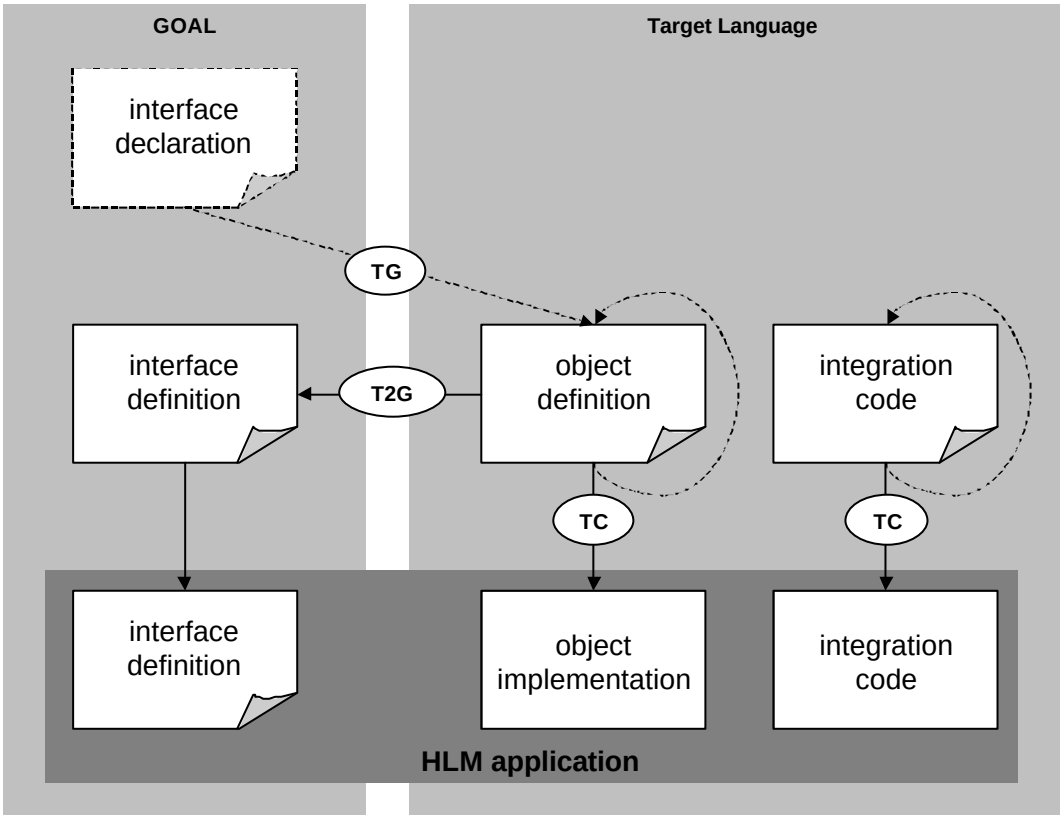

Figure 3.u: The GOAL based development of applications of the HLM migration mechanism uses a GOAL-to-target compiler (G2T) to translate the manually developed GOAL source files into the particular target language (source files are shown here as dog-eared rectangles, manual changes to source files are shown as dashed arrows). The resulting object definitions are compiled together with additional manually developed integration code into an executable with the use of the normal compiler of the target language (TC).

Additionally, all uses of language concepts that are not supported by the HLM migration mechanism will be reported by the target-to-GOAL compiler as an error since GOAL interface definitions can not be generated in this case. This includes for example language specialties like friends of C++ [Str1991] or before and after methods of CLOS [Pae1993].

Optionally, the management of invocation counters for each relevant signature of `OM_Migrateable` interfaces can be checked by the target-to-GOAL compiler statically as an increment has to be inserted as the first statement of each method and each return statement within every method body has to be preceded by a counter decrement regardless of its actual use during execution. Invocation counters for all relevant components of `OM_Owner` interfaces can be checked but the use of source code hints like special comments is advised in order to reduce the checks to relevant cases.

As a side effect the target-to-GOAL compiler will be able to produce a list of the interface-names of the maximum interface set required for each migrateable objects of the corresponding application. The list can be used in additional tests to determine whether objects of the respective interface will be able to migrate between particular environments.

After the development of an application of the HLM migration mechanism need to be tested. Like every programming language tool the HLM migration mechanism can not guarantee that an object can be migrated. Extensive testing is required to make sure the implemented code actually works as intended and a test tool can be helpful in this regard.

Test Tool

A full test of an application of the HLM migration mechanism would require that a migration is attempted for every possible combination of hardware, operating system, programming language, library and application heterogeneity a migrateable object can be confronted with. This is obviously an insurmountable task.

In order to ensure that the migrateable objects can be migrated in real life situations a test tool that is able to simulate the conditions of an actual migration can be very helpful in the debugging of applications of the HLM migration mechanism. In order to reduce the number of test cases to the relevant ones the test tool should be able to load only a defined number of interfaces into the participating environments.

One possible approach to testing object migration would be the simulation of actual migration requests. The amount of interfaces available could be varied and the number of existing objects at the destination environment could be changed. Although its is impossible to test all relevant situations, some extreme as well as various typical conditions for migration can be checked.

Dependent on the problem domain of the particular application certain test scenarios as well as a number of well known environments that are likely targets for a migration can be chosen to "certify" the migrateable objects against. Although this approach does not guarantee migrateability in the general case it can be sufficient for specific purposes (see also page 125).

Integrated Development Environments

Apparently all of the tools described above could be aggregated into the various existing integrated development environments (IDE). Especially the template generator would be very helpful in conjunction with the analysis and design tools that are in common use today. The GOAL-to-target and the target-to-GOAL compilers could be integrated as well and the test tool could be part of distributed debugging environments. The target-to-GOAL compiler for example could be activated automatically in the context of incremental compilation.

A tight integration of the development tools will on the other hand limit the extensibility of the migration mechanism. Changes to the mechanism will have to be reflected within the development tools as well. A tighter integration of these tools will result in additional dependencies that have to be taken into account and might limit innovation.

3.8 Migration in Practice

This subchapter describes the design and implementation of a sample application of the HLM migration mechanism. This application serves both as a proof of concept for the prototypical implementation of the HLM migration mechanism and as a demonstration of the benefits of heterogeneous migration in general.

The sample application uses the HLM migration mechanism to trade derivatives, i.e. financial instruments whose values are derived from other securities or currencies. Migration is used by

the application to close deals where derivatives are sold and bought among brokers of a derivatives marketplace[17].

The application that can be used by a derivatives broker is called *DerivativeShell*. The following subchapters describe some details of the problem domain, the design and the implementation of the DerivativeShell application as well as a protocol of two concurrent sessions that demonstrate the use of the HLM migration mechanism by the DerivativeShell.

3.8.1 Problem Domain

A *derivative* is a financial instrument that is based on an so called *underlying security*. It is issued by a financial institution that grants the holder[18] of the derivative certain rights about the underlying securities, for example to buy shares of the underlying security at a certain price. The derivatives themselves are traded and the price of the derivative (not of the underlying security !) depends on the granted rights and fees of the derivative as well as on the current quotation of the underlying security.

A great number of derivatives exist as derivatives can be defined freely by the issuing financial institutions using different parameter to define the rights and fees. Without loss of generality the scope of this sample application of the HLM migration mechanism will be confined to only two of the most well-known types of derivatives, so called calls and puts.

Call

A *call* is a derivative that entitles its holder to buy shares of the underlying security at a certain price, called the *execution price*. The issuing institution is charging the holder of the derivative a fee called *premium* when the right granted by the call is executed. The call itself is traded freely and its price is usually related to the current quotation of its underlying security.

A broker will buy a call if he expects that the quotation of the underlying security will rise above the execution price of the call. The buyer will then be entitled to buy shares of the underlying security below the quotation. He will be able to sell the shares at a higher price and yield the difference of the execution price and the quotation as earnings after a deduction of the premium as well as the price he has bought the call at. The issuer of the call has to provide the underlying shares that the holder of the call will request if the call is executed.

However, if the quotation of the underlying security does not rise above the execution price of the call, the buyer of the call will not execute the granted right and will loose the money the call was bought at. The issuer of the call on the other side will yield the price he sold the call at as earnings.

A call can be characterized as a bet between the issuer of the call and the holder whether the quotation of the underlying security will rise or not. The issuer thinks its will not rise and offers to bet against it and the buyer of the derivative thinks it will rise and bets for it through buying the call offered.

The issuer of a call can limit his potential losses in two ways. A so called *cap price* as well as an *expiration date* can be defined for a call. A *cap price* is the maximum quotation of the underlying security at which the call will be executed. The cap price of a call is always significantly higher than the execution price of the call.

The cap price limits the amount of money the issuer of the call will have to pay in order to provide the shares for the execution of the call. The cap price also limits the potential earnings of the holder of the call. The expiration date of a call limits the time the call can be traded freely as it defines the day at which a call will be executed irrevocably. Figure 3.v illustrates the parameters of a call.

The price a call is traded at is usually only a fraction of the quotation of the underlying security. The lower the probability that the call will be executed the lower the price will be. The maximum price is theoretically defined by the maximum earnings, i.e. the cap price minus the execution

[17] The description of derivatives has been simplified in order to reduce the complexity of the sample application.

[18] The correct legal term would be “owner”, but the term “holder” is used here in order to avoid confusion with terminology used for the migration architecture.

price minus the price the call was bought at. The initial price of a call is defined by the issuer based on the execution-probability at the time of issue. If a broker is willing to take the bet he will buy the call. The holder of the call may sell the call to another broker at any price.

The value of a call to its holder does not only depend on the quotation of the underlying security but also on the price the call was bought at. Even if the quotation of the underlying security is already above the execution price of the call the value to the holder may still be negative if it was bought at a price higher then the difference between the quotation and the execution price minus the premium.

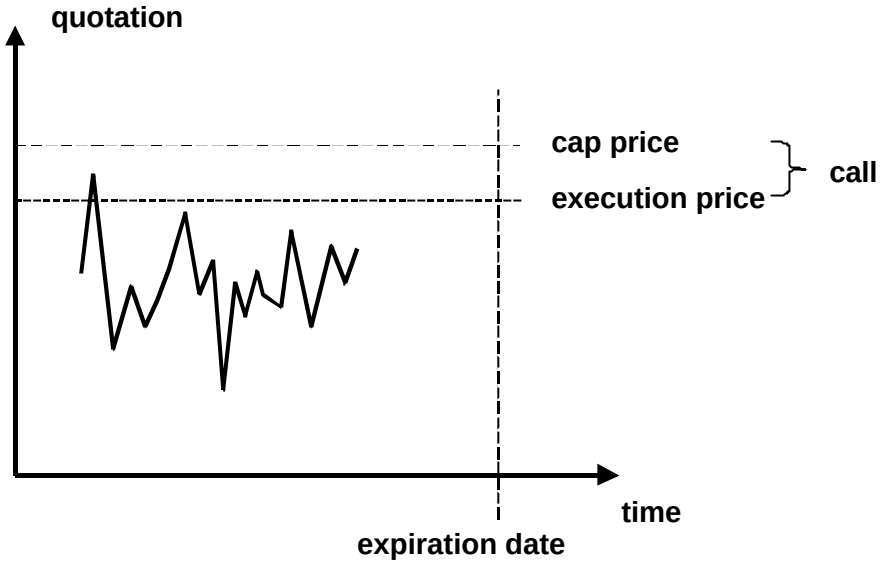

Figure 3.v: A call is defined by an exercise price, a premium (not shown), a cap price and a expiration date. If the quotation of the underlying security rises above the execution price the holder of the call is entitled to buy shares of the underlying security at the execution price. If the quotation of the underlying security reaches the cap price or the expiration date is reached, the call is executed immediately.

Whether a holder of a call will execute the call as soon as its value is positive, whether the holder will wait until the quotation of the underlying security reaches the cap price or until the expiration date is reached, whether the holder will sell the call and at what price depends on the market conditions and the temper of the holder. Apart from its relation to the underlying security a derivative is traded just like any other security.

Put

A put is a derivative that is the opposite from a call. A put entitles the holder to sell shares of the underlying security at the execution price. The issuer of a put is also charging the holder a premium for the execution of the put. A put may also be limited by an expiration date and by a cap price at which the put is executed immediately.

A broker will buy a put if he expects that the quotation of the underlying security will fall below the execution price of the put. The buyer will then be able to buy shares of the underlying security on the free market below the execution price. Through execution of the put he will be able to sell the shares at the higher execution price and yield the difference of the execution price and the quotation as earnings after a deduction of the premium as well as the price he has bought the put at. The issuer of the put has to buy the underlying shares that the holder of the call will offer if the put is executed.

However, if the quotation of the underlying security does not fall below the execution price of the put, the buyer of the put will not execute the granted right and will loose the money the put was bought at. The issuer of the put on the other side will yield the price he sold the put at as earnings.

A put can be characterized as a bet between the issuer of the put and the holder whether the quotation of the underlying security will fall or not. The issuer thinks its will not fall and offers to bet against it and the buyer of the derivative thinks it will fall and bets for it through buying the put offered. The expiration date and the cap price work just as with calls with the exception that the cap price of a put is always significantly lower than the execution price of the put. Figure 3.w illustrates the parameters of a put.

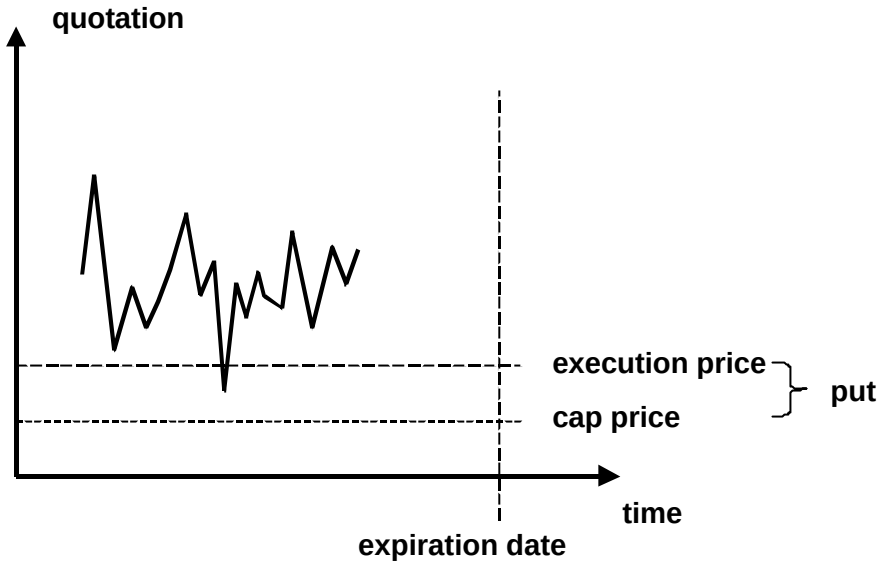

Figure 3.w: A put is defined by an exercise price, a premium (not shown), a cap price and a expiration date. If the quotation of the underlying security falls below the execution price the holder of the put is entitled to sell shares of the underlying security at the execution price. If the quotation of the underlying security reaches the cap price or the expiration date is reached, the put is executed immediately.

Call and puts are comparatively well known among investors as they promise to yield high earnings with the investment of comparatively low money. On the other side, even a small investment will be lost completely if the quotations of the underlying security are moving into the "wrong direction".

A financial institution that issues calls and puts will yield its earnings from the premium that is charged when the calls and puts are executed as well as from the "lost bets", i.e. those calls and puts that have not been executed when they expire and the full issuing price will be gained. A financial institution will try to define the parameters of the call and puts it issues in the way most favorable to its own interests.

The real significance of the derivatives does not stem from their speculative nature but rather from their use as a financial instrument. Derivatives are frequently used by institutional investors to limit potential losses of their investments. Derivatives are also deliberately issues by financial institutions for that purpose.

If for example an institutional investors makes a large investment in a certain security because a long term increase of its quotation is expected the investor may also buy a equal number of puts at only a tiny fraction of the investment. The put will be used as an "insurance" against a short term decline of the quotation as potential losses will be limited. The investor will also gain the time until the puts expires to decide which other securities appear more profitable.

This use of derivatives as an "insurance" against uncertainties is even more formalize through the combination of elementary derivatives to higher order derivatives. Almost arbitrary combination of derivatives can be envisioned but only one of the most well known is described here, the so called spreads.

Higher Order Derivatives: Spreads

A *spread* is a combination of a call and a put with similar parameters to a higher order derivative that can be traded just as any other security. A spread has guarding effect with respect to an underlying security as it limits losses if the quotations falls and it guarantees earnings if the quotation rises. Figure 3.x illustrates the effect of a spread.

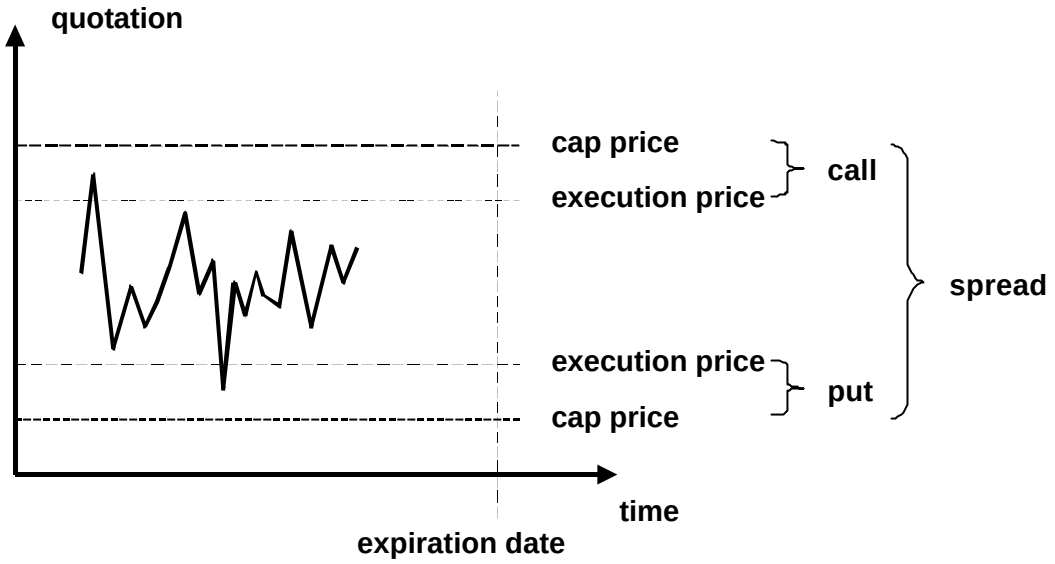

Figure 3.x: A spread is a higher order derivative that combines a call and a put in order to limit potential losses and to guarantee earnings with regard to the underlying security.

Elementary as well as higher order derivatives can be combined by financial investors for arbitrary purposes. One popular use lies in the limitation of currency-risks for investments in foreign markets. If for example the exchange rate Euro-Yen can be balanced against a base currency like the US-Dollar through the combination of two spreads Euro-Dollar and Dollar-Yen.

It is important to note that the value of a higher order derivative to its holder is influenced by the parameters of all derivatives the spread is constructed from as well as from the purchase prices of all participating derivatives. The possibilities to combine simpler derivatives to higher order derivatives are virtually limitless.

3.8.2 Application

The DerivativeShell application of the HLM migration mechanism resembles the computerized workplace of a derivatives broker who is also able to issue new kinds of derivatives as well. The application solves several problems inherent to the volatile derivatives marketplace through the use of the HLM migration mechanism.

Derivatives are comparatively short-lived products and the derivatives marketplace is characterized by a high velocity of trading. The lifespan of derivatives from their inception to their expiration ranges from weeks to month. A newly issued derivative has to be brought into the marketplace very quickly.

The issuer of a new derivative has to have full control over the characteristics of his creation, e.g. under which conditions the derivatives is offered and how the various parameters are used. A broker on the other hand must have full confidence about these characteristics and must be able to compute the value a derivative has correctly.

Both issuer and broker must be able to combine existing derivatives to higher order derivatives in arbitrary ways with full confidence in the correct computation of the value of such higher order derivatives. Both issuer and broker must also be able to sell and buy single derivatives as well as higher order derivatives freely.

Conventional Approach

A conventional approach to support issuers and brokers of derivatives will probably offer both an information systems that implements the computation of the value of the various derivatives. The trading of derivatives will probably be implemented as an exchange of data that defines the various parameters of the derivatives traded.

Although functional, such a conventional approach implies a severe limitation that restrains the full development of a derivatives marketplace. In the context of the conventional approach the rate at which new versions of the supporting software can be deployed among the members of a marketplace limits the rate of innovation.

If an issuer who wants to change the characteristics of a new derivative in a way that is not previously implemented by the supporting software will only be able to offer his creation in the marketplace if a new version of the software is created and distributed among all members of the marketplace. A broker on the other hand will not buy a new derivative if he can not be sure that its value is computed correctly.

This problem will become even more problematic if the supporting software is implemented for several different platforms. The various heterogeneous versions of the software communicate with each other for the trading of derivatives but may be implemented quite differently. In order to introduce a new kind of derivative new versions of the software for each supported platform will have to be created and deployed.

Using the HLM Migration Mechanism

The limitation of the conventional approach can be avoided if the supporting software is implemented using object technology and the HLM migration mechanism is used to exchange not only the parameters of derivatives but also their semantics among heterogeneous platforms and implementations.

Derivatives can be easily implemented as objects that encapsulate not only their parameters as state but also as their behavior how their value is computed and how they interact with other derivates. The HLM migration mechanism will be able to transfer both the state and the semantics of derivatives through migration among heterogeneous systems.

The design of an application that uses the HLM migration mechanism to support the creation and trading of derivatives is quite simple and illustrates the use of the migration architecture. Following the conventions of the HLM migration mechanism the application is implemented as an interface named `OM_DerivativeShell`.

The derivatives are implemented as an interface `OM_Derivative` that inherits from the `OM_Migrateable` interface since derivatives are supposed to be migrated between environments. The interface `OM_DerivativeShell` that manages the `OM_Derivative` objects inherits from the `OM_Owner` interface.

The `OM_DerivativeShell` does not store the derivatives itself, but uses an interface `OM_DerivativeSet` that inherits from the standard interface `OM_MigrateableSet`. The `OM_DerivativeShell` also centrally manages the quotes of the underlying securities through the use of the interfaces `OM_Quote` and `OM_QuoteSet` respectively[19].

The `OM_DerivativeShell` implements a read-eval-print loop that offers the user a command line interface to create, manage and trade derivatives. The user may choose from the following commands:

- **help**

  The help commando displays a summary of the available commands and their parameters in alphabetical order. The description of each command also distinguish required as well as optional parameters.

[19] In the context of this demonstrative application quotations are generated randomly.

- **create**

  The create command enables the creation of a new derivative with the following parameters: the name of the interface that implements the derivative, the name of the derivative and the name of the underlying security. Optionally the execution price, the premium, the cap price, and the expiration date[20] can be specified as well.

- **list**

  The list command displays a list of all derivatives in the portfolio. The derivatives in the portfolio are numbered and higher order derivatives are indicated through a hierarchical numbering using a dot sign a the delimiter. The listing displays the number and the name of the derivative, the name of the underlying security, the execution price, the premium, the cap price, the purchase price, the current quotation of the underlying security, the resulting value of the derivative to the holder and the name of the interface that implements the derivative[21].

- **sub**

  The sub command allows the construction of higher order derivatives through the subsumption of one derivative under the other. Both derivatives are indicated through their number in the listing of the portfolio.

- **sell**

  The sell command can be used to offer a derivative with another broker who is willing to buy the derivative. The derivative to sell is indicated through the number of the derivative in the listing of the portfolio.

- **buy**

  The buy command is used to indicate a willingness to buy a derivative at a certain price from anyone who is selling at that price.

Through these commands the `OM_DerivativeShell` can be used to create derivatives, view their parameters, build higher order derivatives out of existing ones and to sell and buy both single and higher order derivatives. The `OM_DerivativeShell` checks the parameters typed by the user and indicates whether the respective command was executed correctly or not.

The commands of the `OM_DerivativeShell` application offer the minimal functionality that is necessary to support the problem domain. Several additional command may be useful but the application is focused on the functionality necessary to demonstrate the feasibility and the benefits of the HLM migration mechanism.

In addition to the basic characteristics of the problem domain a specific scenario within that domain is used to emphasize the benefits of the HLM migration mechanism.

Scenario

An member of the derivatives marketplace wants to offer new kinds of derivatives whose premium is not a constant but dependent on the development of the underlying security. The premium of one kind of derivative is supposed to be priced at a fixed percentage of the quotation of the underlying security. The premium of a second kind of derivative is supposed to be priced at a percentage of the average quotation of the underlying security since its creation.

In the context of this scenario an issuer implements an interface to be named `OM_Derivative_WithStatistics` that inherits from `OM_Derivative` and is able to accumulate and use statistical information about its underlying security. The statistical data is managed by an interface named `OM_QuoteStatistics`.

---

[20] The expiration date is not used in the demonstration and only included here for completeness.

[21] The name of the interface is added here for demonstrational purposes and would not be necessary in reality.

The issuer also implements an interface named `OM_Derivative_Percent` that inherits from `OM_Derivative_WithStatistics` and computes its premium as a percentage of the quotation of the underlying security at the time of execution. Another interface `OM_Derivative_Average` that also inherits from `OM_Derivative_WithStatistics` and computes its premium as a percentage of the average quotation of the underlying security since the time its was created.

If the different derivatives are combined to higher order derivatives they must be able to share the same statistics. In order to support this additional feature. The issuer also implements a new version of the trading workplace using an interface named `OM_DerivativeShell2` that inherits from `OM_DerivativeShell` and offers an additional command.

- **share**

  The share command allows two derivatives to share common statistics. Both derivatives are indicated through their number in the listing of the portfolio.

The interfaces defined by the issuer are additions to the interfaces defined for the common application that supports derivatives trading marketplaces. Figure 3.y shows the inheritance lattice of the objects the application is constructed of in the context of the migration architecture.

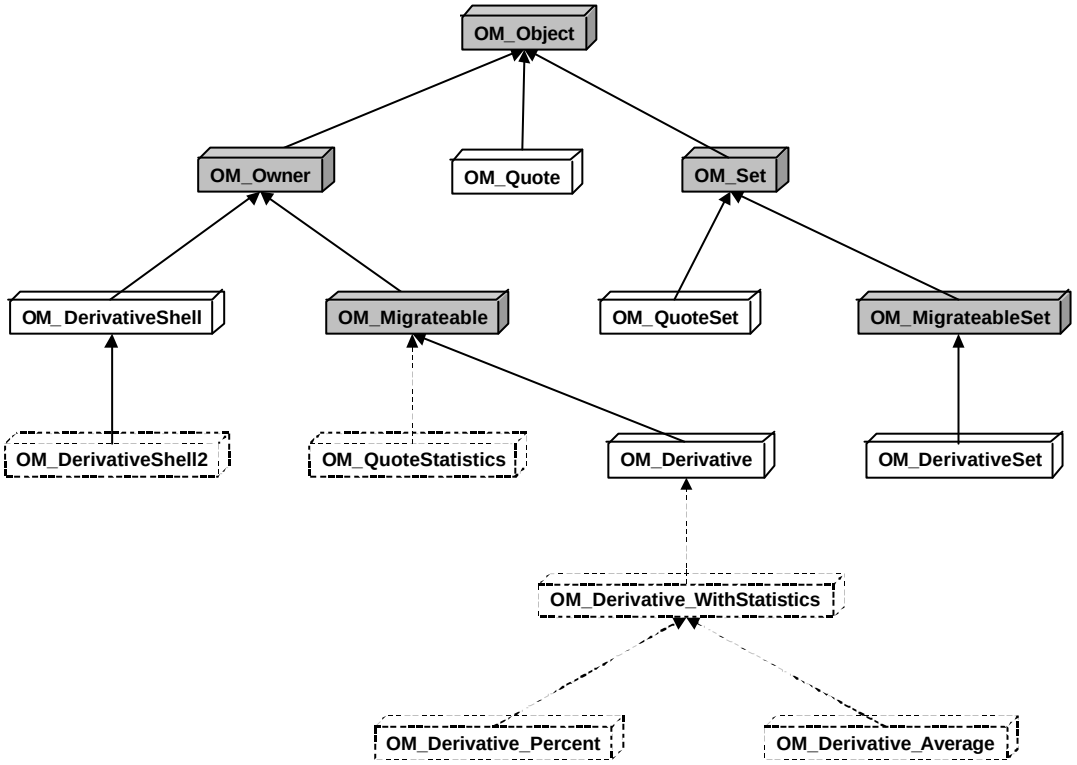

Figure 3.y: The inheritance lattice of the interfaces used by the `OM_DerivativeShell` application in the context of the migration architecture (interfaces that are added for the `OM_DerivativeShell2` application are depicted with dashed lines).

The architecture of the of the application is quite simple. The `OM_DerivativeShell` references a `OM_DerivativeSet` and a `OM_QuoteSet` object in order to manage `OM_Derivative` and `OM_Quote` objects respectively. The managed `OM_Derivative` objects can reference other `OM_Derivative` objects up to an arbitrary depth.

In the context of the `OM_DerivativeShell2` some derivative objects may reference `OM_QuoteStatistics` objects and several derivative objects may share the same `OM_QuoteStatistics` objects as well. Figure 3.z shows the architecture of the application that supports the derivative marketplace in the context of the HLM migration mechanism.

Simplifications

As pointed out earlier the sample applications are focused on the demonstration of the feasibility and the benefits of the HLM migration mechanism in the context of the problem domain. Without loss of generality several simplifications have been applied to the sample applications. These limitations can be addressed comparatively easily in order to make the applications usable in the real world.

The applications support both issuers and brokers of derivatives, but in the real world a broker may not be entitled to issue derivatives. In order to be able to demonstrate different scenarios both roles are supported by the same applications. A separation of these user-roles can nevertheless be implemented easily.

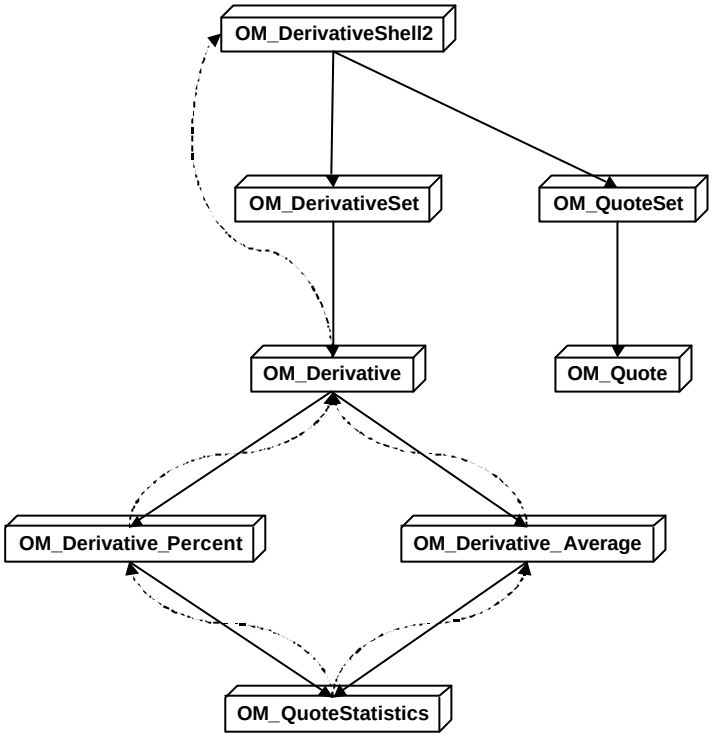

Figure 3.z: The architecture of the application that supports the derivative marketplace. The hierarchy of derivatives objects that construct a higher order derivative can be arbitrarily deep. Statistics objects can be shared by derivatives (the references from migrateable objects to their owners are shown as dashed arrows).

Quotations and all parameters of derivatives are defined as integers which represent basepoints. This limitation was necessary as the prototypical implementation of the HLM migration mechanism does currently not implement floating point operations. It also simplifies the consistent formatting of the display significantly. The same holds for the expiration date which is not displayed. Both support for floating point operations and more sophisticated formatting functionality can be added to the mechanism with reasonable effort.

The quotations of underlying securities are only simulated through a random number generator, primarily in order to be able to demonstrate the application offline. The applications do not offer a mechanism for price-finding, price-negotiation or for the monetary transactions that are necessary to buy derivatives in the real world. The trading of derivatives is also reduced to the simplest case, i.e. a single derivative is traded for a single share of an underlying security. In the real world a derivative may entitle rights for many shares of the underlying security and many derivatives may be traded at once.

Last not least several derivatives marketplaces exist that have specified different conditions for the derivatives that can be offered and traded. The sample applications are designed to trade derivatives freely but can also be extended to obey the standards of different real world derivatives marketplaces.

3.8.3 Demonstration

The following demonstration shows the feasibility of the HLM migration mechanism and its benefits in the context of the application scenario described in the previous subchapter. The demonstration is based on the prototypical implementation of the HLM migration mechanism for the Java and CLOS language environments.

The demonstration is documented through a transcripts of the interactions of two users with their respective applications. An issuer of a derivative marketplace uses a CLOS based version of the `OM_DerivativeShell2` application to create and offer a higher order derivative that is composed out of three elementary ones. A broker of that marketplace uses the `OM_DerivativeShell` application to buy the newly issued derivative.

The transcripts of interactions of the users are shown side-by-side in a two-page format. The interactions are broken up into several steps and comments are inserted to explain what is happening. The interactions of the issuer that acts as the source of the resulting migration are shown on the left hand side and the interactions of the broker that acts as the destination of the resulting migration are shown on the opposite page on the right.

The transcript shows the interactions, i.e. the input/output of the applications in a **`bold fixed-width font`**. Comments are inserted using the normal font. The layout of the transcript "synchronizes" the sequence of events through the insertion of the appropriate number of blank lines of either side. These blank lines to not appear in the real interactions.

Source Transcript

```
OM_DerivativeShell2 started !
OM_DerivativeShell2>
```

The `OM_DerivativeShell2` is started and displays its command line prompt.

```
OM_DerivativeShell2>
command: create OM_Derivative_Percent Call ABC 600 5 750
Created derivative:
#      Name   USec   ExecP  Prem   CapP   PurP   Quote  Value  Interface
1      Call   ABC    600    34(5)  750    0      679    45     OM_Derivative_Percent
              min:   679    avg:   679    max:   679
```

A first derivative is created using the `OM_Derivative_Percent` interface. It is named `Call`, is based on the underlying security ABC, has an execution price of 600 basepoints, a premium of 5 percent and a cap price of 750 base points. The output immediately after the creation shows the parameters of the derivative, a purchase price of 0, a quote of 679 and a resulting value of 75. The value of the derivative is the difference of the quotation an the execution price minus the premium which is 34 basepoints (5% of the quotation). The line below the listing of the derivative shows the data gathered by the automatically created `OM_QuoteStatistics` object so far, i.e. minimum, average and maximum quotations of the underlying security ABC.

```
OM_DerivativeShell2>
command: create OM_Derivative_Average Put ABC 400 10 250
Created derivative:
#      Name   USec   ExecP  Prem   CapP   PurP   Quote  Value  Interface
2      Put    ABC    400    0(10)  250    0      665    0      OM_Derivative_Average
              min:   665    avg:   665    max:   665
```

A second derivative is created using the `OM_Derivative_Average` interface. It is named `Put`, is based on the underlying security ABC, has an execution price of 400 basepoints, a premium of 10 percent and a cap price of 250 base points. The output immediately after the creation shows the parameters of the derivative, a purchase price of 0, a quote of 665 and a resulting value of 0. The value of the derivative is 0 because the quotation is higher then the execution price, i.e. the derivative can not be executed. The line below the listing of the derivative shows the data gathered by the automatically created `OM_QuoteStatistics` object so far, i.e. minimum, average and maximum quotations of the underlying security ABC.

Destination Transcript

```
OM_DerivativeShell started !
OM_DerivativeShell>
```

The `OM_DerivativeShell` is started and displays its command line prompt.

```
command: list
Derivatives in the portfolio:
#       Name    USec    ExecP   Prem    CapP    PurP    Quote   Value   Interface
none
Legend:
USec - Underlying Security, ExecP - Execution Price, Prem – Premium
CapP - Cap Price, PurP - Purchase Price
```

The list command is executed and shows an empty portfolio of derivatives.

Source Transcript (continued)

```
OM_DerivativeShell2>
command: share 1 2
Share operation successful !
```

The share command is executed to share the `OM_QuoteStatistics` object of derivative number 2 with derivative number 1, the other `OM_QuoteStatistics` Object is discarded.

```
OM_DerivativeShell2>
command: list
Derivatives in the portfolio:
#      Name   USec   ExecP  Prem   CapP   PurP   Quote  Value  Interface
1      Call   ABC    600    45(5)  750    0      671    26     OM_Derivative_Percent
              min:   665    avg:   668    max:   671
2      Put    ABC    400    67(10) 250    0      671    0      OM_Derivative_Average
              min:   665    avg:   668    max:   671
Legend:
USec - Underlying Security, ExecP - Execution Price, Prem – Premium
CapP - Cap Price, PurP - Purchase Price
```

The list command is executed and displays the two derivatives in the portfolio which now share a single `OM_QuoteStatistics` object (indicated by the identical statistics).

```
OM_DerivativeShell2>
command: create OM_Derivative Spread ABC
Created derivative:
#      Name   USec   ExecP  Prem   CapP   PurP   Quote  Value  Interface
3      Spread ABC    0      0      0      0      669    0      OM_Derivative
```

A third derivative is created using the `OM_Derivative` interface. It is named `Spread` and is based on the underlying security ABC but no optional parameters are used.

```
OM_DerivativeShell2>
command: sub 3 1
Subsume operation successful !

OM_DerivativeShell2>
command: sub 2 1
Subsume operation successful !
```

The sub command is executed two times to subsume the two previously created derivatives under the last one to create a higher order derivative.

```
OM_DerivativeShell2>
command: list
Derivatives in the portfolio:
#      Name   USec   ExecP  Prem   CapP   PurP   Quote  Value  Interface
1      Spread ABC    0      0      0      0      673    40     OM_Derivative
1.1    Call   ABC    600    33(5)  750    0      673    40     OM_Derivative_Percent
              min:   665    avg:   670    max:   673
1.2    Put    ABC    400    67(10) 250    0      673    0      OM_Derivative_Average
              min:   665    avg:   670    max:   673
Legend:
USec - Underlying Security, ExecP - Execution price, Prem – Premium
CapP - Cap Price, PurP - Purchase Price
```

The list command displays the higher order derivative with the subsumed derivatives indicated with hierarchical numbering. The value of the higher order derivative is computed as the sum of the subsumed derivatives.

Destination Transcript (continued)

```
OM_DerivativeShell>
command: buy Spread ABC 50 9876
Waiting for offer at port: 9876
```

The buy command is used to indicate the willingness of the user to buy a derivative with the name Spread of the underlying security ABC for the amount of 50 basepoints using the TCP/IP port 9876.

The following output is generated solely for the demonstration of the HLM migration mechanism.

```
OM_Porter.handle_migration_via_network() begin
listening at: 9876
```

As a consequence the `handle_migration_via_network()` signature of the `OM_Porter` object is invoked and starts to listen at TCP/IP port 9876 for incoming communications from other environments.

Source Transcript (continued)

```
OM_DerivativeShell2>
command: sell 1 9876 127.0.0.1
Selling derivative: 1 at port: 9876 to host: 127.0.0.1
```

The sell command is used to initiate the sale of the derivative number 1 via port 9876 at TCP/IP address 127.0.0.1.

The following output is generated solely for the demonstration of the HLM migration mechanism.

```
OM_Porter.migrate_via_network() begin

INITIATION


connection established
```

The `migrate_via_network()` signature of the `OM_Porter` objects is invoked to perform the migration of the derivative object.

The initiation phase successfully establishes a connection with the `OM_Porter` object of the destination environment.

```
CHECK
ready to migrate
migration set:
OM_Derivative
OM_Derivative_Percent
OM_QuoteStatistics
OM_Derivative_Average
```

The check phase performs the migration check and displays the interfaces of the objects of the migration set.

```
NEGOTIATION
requesting supported interfaces



supported interfaces:
OM_DerivativeShell
OM_Character
OM_DerivativeSet
GOAL_Token
OM_Directory
OM_Environment
OM_Error
OM_File
OM_Migrateable
OM_MigrateableSet
OM_Owner
OM_OwnerSet
OM_Porter
OM_ServerSocket
OM_Set
OM_Socket
OM_Stream
OM_String
OM_Token
OM_Quote
OM_QuoteSet
OM_Object
OM_Interface
OM_InterfaceSet
OM_Derivative
```

The negotiation phase requests the supported interfaces from the `OM_Porter` of the destination environment and displays the list upon receipt.

Destination Transcript (continued)

```
connection established !
```

The `OM_Porter` object of the destination environment acknowledges the communication with the source environment.

```
command: SEND_SI
supported interfaces sent !
```

The `OM_Porter` object of the destination environment receives the command to send the supported interfaces to the source environment.

A confirmation is printed that all interface declarations have been sent.

Source Transcript (continued)

```
collecting interfaces to process

interfaces to process:
OM_Derivative
OM_Derivative_Percent
OM_QuoteStatistics
OM_Derivative_Average
```

The set of interfaces to process, i.e. the negotiation set is constructed as a collection of the interfaces of the objects of the migration set.

```
processing interfaces

interface set:
OM_Derivative_Percent
OM_QuoteStatistics
OM_Derivative_Average
OM_Derivative_WithStatistics
```

The interface set is determined as the set of dependent interfaces that are not supported by the destination environment.

```
TRANSFER OF SEMANTICS

interfaces sent
```

The transfer of semantics phase generates an interface definitions for each interface in the interface set and transfer that representation to the `OM_Porter` object of the destination environment.

A confirmation is printed when all interface definition have been sent.

```
interfaces implemented
```

The `OM_Porter` object acknowledges the implementation of the interface definitions by the destination environment.

```
TRANSFER OF STATE
objects to represent:
OM_Derivative
OM_Derivative_Percent
OM_QuoteStatistics
OM_Derivative_Average

representation:
OM_Derivative 1 OM_Derivative_Percent 2 OM_QuoteStatistics 3 OM_Derivative_Average 4
| 1 (Spread ABC 0 0 0 :2 :4 ) 2 (Call ABC 10 600 750 :3 ) 3 (ABC 673 665 2009 3 ) 4
(Put ABC 10 400 250 :3 )

local objects deactivated: ready to commit locally

representation sent
```

The transfer of state phase constructs the ORL representation of the objects of the migration set, prints that representation, and deactivates the object of the migration set, to reach “ready to commit” state.

A confirmation is printed when the ORL representation has been sent to the `OM_Porter` object of the destination environment.

```
objects represented: ready to commit remotely
```

The `OM_Porter` object acknowledges the representation of the objects of the migration set by the destination environment.

Destination Transcript (continued)

```
command: IMPLEMENT

javac -d c:\classes c:\src\goal\om\dest\OM_Derivative_Percent5.java
javac -d c:\classes c:\src\goal\om\dest\OM_QuoteStatistics.java
javac -d c:\classes c:\src\goal\om\dest\OM_Derivative_Average10.java
javac -d c:\classes c:\src\goal\om\dest\OM_Derivative_WithStatistics.java

interfaces implemented
```

The `OM_Porter` object of the destination environment receives the command to implement the interface definitions that are transferred by the `OM_Porter` object of the source environment. Local source files are generated and complied for each interface received.

A confirmation is printed and sent to the `OM_Porter` object of the source environment.

```
command: REPRESENT

OM_Derivative 1 OM_Derivative_Percent5 2 OM_QuoteStatistics 3 OM_Derivative_Average10
4  | 1 (Spread ABC 0 0 0 :2 :4 ) 2 (Call ABC 10 600 750 :3 ) 3 (ABC 673 665 2009 3 ) 4
(Put ABC 10 400 250 :3 )

objects represented
```

The `OM_Porter` object of the destination environment receives the command to represent the objects of the migration set and prints the ORL representation received.

A confirmation is printed and sent to the `OM_Porter` of the source environment when all objects of the migration set have been successfully rebuild and reinitialized.

Source Transcript (continued)

```
COMMIT
```

The commit phase sends a commit command to the `OM_Porter` of the destination environment.

```
objects activated remotely
```

An acknowledges of the activation of the objects of the migration set by the destination environment is printed.

```
objects released locally
migration successfull
```

The `OM_Porter` object releases the objects of the migration set and prints an acknowledgement of the success of the migration operation.

```
closing connection
```

The connection with the destination environment is closed.

```
OM_Porter.migrate_via_network() end
```

The invocation of the `migrate_via_network()` signature returns.

The following output represents the normal interaction of the `OM_DerivativeShell2` with its user that is independent of the destinations environment.

```
Sell operation successful !
```

The sell command of the `OM_DerivativeShell2` is finished.

```
OM_DerivativeShell2>
command: list
Derivatives in the portfolio:
#      Name   USec   ExecP  Prem   CapP   PurP   Quote  Value  Interface
none
Legend:
USec - Underlying Security, ExecP - Execution price, Prem – Premium
CapP - Cap Price, PurP - Purchase Price
```

The list command is invoked and displays an empty portfolio of derivatives.

End of Transcript

Destination Transcript (continued)

```
command: ACTIVATE
migrated objects activated
migration successfull
```

The `OM_Porter` object of the destination environment receives the command to commit the migration and activates the objects of the migration set.

An acknowledgement of the activation of the objects of the migration set is printed and sent to the `OM_Porter` object of the source environment.

An acknowledgement of the successful migration set is printed.

```
closing connection
OM_Porter.handle_migration_via_network() end
```

An acknowledgement of the closing of the connection is printed and the invocation of the `handle_migration_via_network()` signature of the `OM_Porter` object returns.

The following output represents the normal interaction of the `OM_DerivativeShell2` with its user that is independent of the source environment.

```
Buy operation successful !
```

The buy command of the `OM_DerivativeShell` is finished.

```
OM_DerivativeShell>
command: list
Derivatives in the portfolio:
#      Name   Usec   ExecP  Prem   CapP   PurP   Quote  Value  Interface
1      Spread ABC    0      0      0      50     677    -6     OM_Derivative
1.1    Call   ABC    600    33(5)  750    0      677    44     OM_Derivative_Percent
              min:   665    avg:   671    max:   677
1.2    Put    ABC    400    67(10) 250    0      677    0      OM_Derivative_Average
              min:   665    avg:   671    max:   677
Legend:
USec - Underlying Security, ExecP - Execution price, Prem – Premium
CapP - Cap Price, PurP - Purchase Price
```

The list command is invoked and displays the newly bought derivative whose value is now computed as –6. The Call and the Put have a combined value of 44 but the Spread was purchased at 50 basepoints resulting in a negative value of the higher order derivative that will change if the quotation of the underlying security rises some more.

End of Transcript

After the end of the transcript, the interactions of the users may continue on both sides and may include additional communication partners as well as more complex higher order derivatives or derivatives of interfaces with more functionality. The HLM migration mechanism will enable the trading of all these derivatives as long as they are implemented in accordance with the migration architecture.

Evaluation

The above transcript demonstrates the feasibility of the HLM migration mechanism and its applicability to the problem domain. The use of the HLM migration mechanism enables the transfer of the derivative objects including their semantics which assures the seller that the value of the bought derivative is computed correctly.

Despite the use of new functionality for the creation of the offered derivative the buying transaction can be performed at runtime without the deployment of new software as the destination environment is extended with the necessary semantics on the fly. The state necessary to compute the parameters of the higher order derivative is also transferred correctly.

Apart from these benefits of the HLM migration mechanism for the problem domain several highlights of this demonstration that are of general interest have to be emphasized:

- **Support for Dynamically Created Objects**

  The HLM migration mechanism is able to handle objects that are created at runtime dynamically, i.e. from interfaces that are not specified in the source code of the source application but that are defined through user input. The `create` commando of the `OM_DerivativeShell` takes the name of the interface to use as a parameter. If the interface is not available an error message will be given.

  This is an important detail of the HLM migration mechanism as the determination of the interface set by the negotiation phase would not be correct if it was based on the object definition of the root object of the migration request alone. The collection of the interfaces of the objects in the migration set ensures that the interfaces of the actual interfaces are investigated not only those specified by the source code of the root object.

- **Support for Arbitrary Object Structures**

  The HLM migration mechanism is able to migrate arbitrary object structures. The higher order derivatives form a hierarchy but the `OM_DerivativeShell2` can be used to build arbitrary objects structures. The `OM_Derivative_With_Statistics` objects as well as the `OM_QuoteStatistics` objects can have arbitrary numbers of components or owners respectively.

  If used independent of their role within the sample application these interfaces can be used to build arbitrary complex graphs of objects that can be migrated by the HLM migration mechanism. Such structures will probably not make much sense in the context of the problem domain and their listings in the `OM_DerivativeShell2` will be virtually unreadable but the ability of the HLM migration mechanism to migrate these structures is a proof of its universal applicability in this regard.

- **Correct Handling of Ambiguous Dependencies**

  The HLM migration mechanism is also able to handle ambiguous dependencies among migrateable objects correctly. Although not visible directly in the above transcript, the `OM_Derivative_Percent` object does not depend on the `OM_QuoteStatistics` object it references and sends it a `ready_to_let_owner_migrate()` message.

  The `OM_Derivative_Average` object on the other side does depend on the `OM_QuoteStatistics` object and sends it a `ready_to_migrate()` message. As a result the `OM_QuoteStatistics` object becomes part of the migration set and that fact is handled correctly by the `OM_Derivative_Percent` object during the construction of its ORL representation as well as during the rebuild operation.

  It is essential to note that the HLM migration mechanism is able to handle ambiguous dependencies of objects correctly as the probability of ambiguities will rise with the number of objects that are involved in a single migration. Ambiguous dependencies are supported though the ORL representation and the relevant interfaces need to be implemented correctly which can be assured through appropriate development tools (see also page 121).

- **Migration of Semantics without Corresponding User Interface**

  It is interesting to note that the above demonstration migrates an object structure to an application that can not be used to create that object structure itself. The `OM_DerivativeShell` application does not support the `sub` command necessary to combine the statistics of two `OM_Derivative_WithStatistics` objects but it is able to receive such a structure through migration although is was designed to support such a structure explicitly.

  Yet the migration makes sense as the migrated objects structure can be displayed like any other higher order derivative and can even be sold, i.e. migrated to other `OM_DerivativeShell` applications without loss of fidelity. This simple example shows that the migration of objects can be useful even between applications that do not have the same user interface.

- **Support for all Kinds of Heterogeneity**

  Last not least, the above demonstration shows that the HLM migration mechanism is able to address all levels of heterogeneity as different hardware, operating systems and language environments are used for the demonstration. The source is a CLOS environment under Microsoft Windows NT running on an Intel PC and the destination is a Java environment running under Solaris of a Sparc workstation. The library level is heterogeneous in so far as different sets of interfaces are used, which could also build two different inheritance hierarchies. An the applications are heterogeneous as they offer different functionality and implement different user interfaces.

  Although the extent of the heterogeneity at the library and application level is not to far the basic feasibility of the HLM migration mechanism is shown. Based on the interfaces of the objects of the migration set the `OM_Derivative_WithStatistics` object is determined as a dependent interface that need to be transferred to the destination environment. An arbitrary number of application as well as library specific interfaces could be determined as well and would be transferred in the same way.

The above demonstration shows that the HLM migration mechanism conforms to its objectives and fulfills its the research goals. It is able to address all levels of heterogeneity, to transfer objects without applying changes to their definitions during migration and is able to work with existing language environments. An in depth analysis of the characteristics of the HLM migration mechanism will be given in the following chapter four that also discusses possible augmentations of the mechanism and the general applicability of heterogeneous language migration.

4 Augmentation of Heterogeneous Language Migration

The HLM migration mechanism implements object migration among heterogeneous language environments with the main objectives not to require changes to object definitions during migration or to the implementation of participating language environments prior to migration. These and other objectives chosen for the HLM migration mechanism are intended to ensure its applicability to existing language environments.

The prototypical implementation the HLM migration mechanism is limited in its applicability by a number of deficiencies but also by its objectives. It is confined to certain characteristics of objects that can be migrated, to the kind of information that can be transferred as well as to a defined set language concepts that are supported.

To overcome these limitations several improvements of the prototypical implementation can be conceived and the HLM migration mechanism can be augmented in several ways. The possible changes to the HLM migration mechanism differ in their scope and consequences and can be classified as enhancements or extensions.

Enhancements to the HLM migration mechanism can be applied without violating the objectives while *extensions* can only be implemented through changes to the mechanism that transgress the constraints implied by the objectives. Figure 4.a illustrates the difference between enhancements and extensions.

Enhancements as well as extensions may widen the applicability of the HLM migration mechanism to additional kinds of objects, new language concepts or even previously unsupported categories of language environments. However several augmentations may be contradictory and can not be implemented together within a single environment. As a consequence a partitioning of the migrateability of objects among environments into disjunctive sets of environments that implement different versions of the migration mechanism may result.

This chapter tries to explore the scope of applicability of heterogeneous migration through the discussion of the most promising enhancements and extensions to the HLM migration mechanism. A brief characterization of the HLM migration mechanism is provided as a starting point and as a reference for the various changes that can be applied.

Enhancements for consecutive migrations, additional standard interfaces, migrations of objects that use non-standard interfaces as well as migrations in the context of distributed identity are discussed in the second subchapter including selected issues of the implementation of these enhancements.

The third subchapter introduces advanced migration techniques that deviate from the objectives of the HLM migration mechanism and that can be used to implement various extensions that broaden the applicability of the mechanism. Techniques that change interface definitions can be used to implement additional forms of negotiation as well as the migration of fragments of objects. Changes to the participating language environment are required to implement the transfer of computations in the context of the HLM migration mechanism.

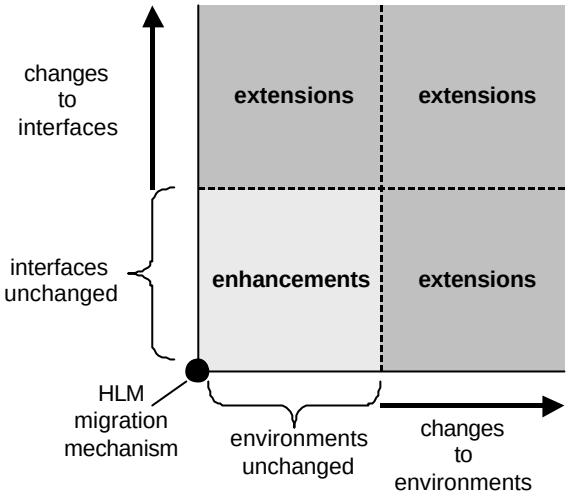

Figure 4.a: The HLM migration mechanism can be augmented through enhancements that adhere to its objectives or through extensions that are not confined by the restrictions implied by the objectives. The above diagram uses the main objectives not to change objects during migration and environments prior to migration as examples.

A discussion of additional language concepts and how they can be supported through the implementation of extensions of the HLM migration mechanism are provided in the fourth subchapter that concludes with an overview of potential migrateability among existing object-based language environments.

4.1 Characterization of the HLM Migration Mechanism

The HLM migration mechanism can be characterized in terms of the systematic introduced in the second chapter. The following characterization of the mechanism discusses each criteria separately and provides some hints at possible enhancements and extensions that will be described in consecutive subchapters.

Unit of Migration

The unit of migration used by the HLM migration mechanism and the level at which migration is performed are obviously atomic objects and the language level respectively. The relative identity and the state of an object to be migrated and its set of dependent objects is transferred. The functionality that is necessary for the these objects to operate within the destination environment is determined and transferred as well. The HLM migration mechanism does not support the transfer of distributed identity or computations of objects.

Identity

The transfer of the distributed identity of objects is not supported by the HLM migration mechanism as it is intended to work with existing language environments most of which do not support distributed identity. Only the relative identity of objects with regard to the set of dependent objects is transferred.

The implementation of location independence through remote references directly will not be possible without changes to the participating environments. Support for distributed identity can however be added to the HLM migration mechanism as an enhancement using proxy objects which unfortunately introduce residual dependencies (see page 158).

State

The state of objects is transferred by the HLM migration mechanism using a source representation in the form of the ORL language. Following a hybrid approach, booleans, integers, floating point numbers, characters and strings are treated as atomic values rather than as objects. The relationships between transferred objects are preserved .

The HLM migration mechanism follows the conversion approach to overcome heterogeneity of hardware and languages as the state of objects of the source environment is converted to an independent representation for the transfer, which is used to rebuild newly created objects within the destination environment. The restriction approach to overcome heterogeneity is applied as well, as the HLM mechanism is confined to objects with mutual relationships.

Alternatively representations of state can be used to enhance the HLM migration mechanism as for example the Extensible Markup Language (XML) defined by the World Wide Web Consortium (W³C) which provides a more versatile textual format. Binary representations as used by persistence services can be employed as long as they support heterogeneity of hardware, operating systems and languages.

Functionality

The transfer of the semantics of objects can be considered the main focus of the HLM migration mechanism and is performed using the GOAL abstraction language. A negotiation algorithm determines which functionality needs to be transferred and ensures that the required functionality is available within the destination environment.

The HLM migration mechanism employs a combination of the conversion and the extension approach to overcome all levels of heterogeneity in this regard. The functionality of objects is converted to an independent representation and the destination environment is extended with object definitions that are generated from the transferred GOAL interface definitions.

The negotiation algorithm can be enhanced through the use of different interface equivalence relations which can widen the applicability of the mechanism but may also lead to only partial migrateability. Extensions of the HLM migration mechanism will enable changes to the functionality of objects for various purposes following the adaptation approach to overcome heterogeneity (see page 174).

The use of intermediate, virtual or binary representations of functionality will only be possible in the form of extensions to the HLM migration mechanism as changes to the participating environments will be necessary. These alternative formats can be used to achieve higher performance as only a partial translation into the native code of the destination platform will be necessary. However the implementation of the necessary conversions will also be more difficult (see page 179).

Computations

The transfer of computations of objects is not supported by the HLM migration mechanism as only few language environments provide means to capture or recreate the state of execution. It is difficult to transfer computations among two different hardware architectures, but the task becomes even harder if it is pursued in the context of heterogeneity of languages, operating systems, and hardware.

The HLM migration mechanism can be extended to transfer computations of objects through the analysis of the execution state of the runtime environment at the source and the reconstruction of an equivalent execution state within the destination environment. Such a fundamental alteration of the mechanism requires changes to the participating environments (see page 179).

Policy

The HLM migration mechanism implements no particular policy for the initiation of migrations. Migration request can be created by any policy that uses the mechanism. Reasons to migrate an object among heterogeneous environments can include the whole spectrum of motivations from load balancing to fault tolerance. The capability of the HLM migration mechanism to add new functionality to existing systems can be regarded as a reason in its own right and can encourage the interactive initiation of object migration as well (see also chapter 3 page 125).

The prototypical implementation of the HLM migration mechanism requires that the destination of a migration has to be specified explicitly in the form of the IP address and port number of a listening `OM_Porter` object. With the possible extension of the HLM migration mechanism to

support distributed identity a reference to a remote `OM_Owner` object may also be used to specify the target or a migration in a less implementation dependent way (see page 158).

Mechanism

The prototypical implementation of the HLM migration mechanism has only minimal prerequisites and does deliberately not use any supportive features of the participating language environments in order to avoid implementational dependencies. The mechanism uses source code as the abstract representations for the transport of state and functionality of objects and employs a migration algorithm with six phases.

Prerequisites

The prerequisite of the prototypical implementation of the HLM migration mechanism are the TCP/IP network protocol and the availability of a listening `OM_Porter` object within the destination environment as well as the ability of the destination environment to load object definitions and create objects dynamically at runtime.

The destination environment has to implement at least the receiving functionality of the HLM migration mechanism which requires also the other prerequisites mentioned. In contrast to the high availability of the TCP/IP network protocol, the ability to load object definitions at runtime or to create objects dynamically is comparatively rare among language environments (see also page 211).

Support

The TCP/IP network transport is the only kind of support of the participating environments used by the prototypical implementation of the HLM migration mechanism. The TCP/IP protocol is used for the communications between the `OM_Porter` objects of the source and destination environments. A distributed file system service can be employed as an alternative transport but its use will also reduce performance.

As an additional kind of support, that is available only within particular language environments, runtime type information systems as well as reflective capabilities can be used to simplify the implementation of the HLM migration mechanism especially throughout the negotiation phase (see also 206).

Abstractions

The HLM migration mechanism uses source code as an abstraction for the transfer of state and the functionality of objects in the form of the ORL and the GOAL languages respectively. Although certainly not an optimal form of representation, the source code abstraction helps to overcome heterogeneity through the compilation of the required functionality.

Despite its use of source code for communication means the HLM migration mechanism makes no assumption about the representation of the interface definitions within the participating environments. Apart from the use of source files by the prototypical implementation other forms for interface definitions can be used as well (see also 206).

The use of reflection that renders the behavior of objects available through for example class objects in CLOS [Pae1993] or Smalltalk [GoR1983], may offer more efficient ways of implementing the negotiation algorithm in certain environments. Other form of representation like virtual machine abstractions can also be adopted if the HLM migration mechanism is extended accordingly (see also page 179).

Algorithm

The HLM migration mechanism consists of the phases initiation, check, negotiation, transfer of semantics, transfer of state and completion. The initiation phase ensures communication with the destination environment. The check phase determines whether the objects in question can be migrated and constructs the migration set, i.e. the set of dependent objects that are migrated collectively.

The negotiation set determines the interface set, i.e. the set of interfaces the object of the migration set are dependent upon but that are not supported by the destination environment.

The transfer of semantics phase implements the interfaces of the interface set while the transfer of state phase reconstructs the objects of the migration set. The completion finally performs the commit of the migration operation.

The migration algorithm of the HLM migration mechanism has not been optimized for performance but can be improved through caching techniques. The migration algorithm will also be affected by enhancements and extensions of the migration mechanism some of which require changes to individual phases while others require fundamental changes (see page 152).

The implementation of support for distributed identity will affect the check as well as the transfer of state phase (see page 158). The negotiation phase can be enhanced through different forms of equivalence relations of interfaces (see page 174). More fundamental extensions to the migration algorithm like the pursuit of the adaptation approach to overcome heterogeneity will change the migration algorithm significantly (see page 177).

Properties

The HLM migration mechanism is working non-preemptively since migration can only be initiated through synchronous message passing. As an additional precondition migration can only be performed when the objects involved are inactive, i.e. their methods have not been invoked and these objects are not referenced by activation records. The migration of computations of objects can only be implemented through an extension of the mechanism (see page 179).

The migration itself is performed atomically as either the object to be migrated and its dependent objects are transferred to the destination as a whole or no changes to the participating environments are made at all. The migration protocol concludes with either the release / activation or the reactivation / abort sequences of a two phase commit protocol that are able to commit or roll back changes made to either environment.

The HLM migration mechanism leaves no residual dependencies at the source as objects are migrated to the destination environment completely. This property may change if support for distributed identity is added to the mechanism. The use of proxy objects will create residual dependencies (see page 158).

Fault tolerance is supported by the HLM migration mechanism to the extent that the impossibility of migration can be detected early in the process and communication failures as well as late aborts of the migration process are handled gracefully[22]. The continuation of a previously interrupted migration, due to for example a communication failure is not supported and the migration will have to be initiated anew by the particular application.

The fault tolerance capabilities of the HLM migration mechanism with respect to migrateability can be improved with the utilization of advanced migration techniques. For example a migration that fails using the standard negotiation may be successful using an advanced form of negotiation (see page 177).

The HLM migration mechanism can be characterized as symmetric as well as transitive with regard to the sequence of migrations. It is symmetric because both the environments involved are able to perform a migration using the migration architecture and the necessary interfaces for the migrated objects will be available within both environments after a migration has been performed. The transitive nature of the migration mechanism stems from the same reasons.

The characterization of symmetry and transitivity does not hold in the context of the distributed identity of an object as only relative identity is maintained during migration. For example an object that is migrated to another environment will not be regarded as the same object when it is migrated back to its original source environment [Sil1996]. Enhancement as well as extension to the migration mechanism may also impede the symmetric as well as the transitive character of the HLM migration mechanism (see page 158).

[22] The handling of communication failures depends on the participating language environments and operating systems.

Levels of Heterogeneity

The HLM migration mechanism addresses all levels of heterogeneity albeit through the use of different means. The heterogeneity of hardware, operating systems and to some extent languages is overcome using common source code representations as abstractions for both the transfer of behavior and state. Not all differences between language environments can be overcome because the HLM migration mechanism supports only a limited set of common concepts that are to be used for the objects to be migrated.

The differences between the programming interfaces of the language environments are addressed to the extent that a common set of standard interfaces is used. Differences of the libraries and applications involved are handled through the negotiation algorithm that identifies those interface definitions that are not supported by the destination environment but are required by the object to be migrated.

Approach to Heterogeneity

The approaches used in the HLM migration mechanism to overcome heterogeneity are restricting, enabling, conversion and extension. The prerequisites of the migration mechanism is restricting the language environments that can participate in migration. The set of supported concepts, the standard interfaces, and the requirements for the design of applications are restricting the definitions of objects that can be migrated.

The implementation of the HLM migration mechanism as well as of the standard interfaces is enabling language environments to participate in migration. The use of abstractions to transfer interface definitions and the state of objects requires conversion between the common representation and the native representations of the participation environments. The interface definitions transferred provide an extension to the destination environment that adds the functionality required by the objects to be migrated.

Some of the restrictions of the HLM migration mechanism can be lifted if additional standard interfaces (see page 154) as well as concepts (see page 189) are supported through enhancements and extensions of the mechanism respectively. The use of different forms of abstractions like binary, virtual or intermediate representations will also require conversion (see page 179). Other approaches to overcome heterogeneity like adaptation are used in the context of extensions to the HLM migration mechanism (see page 174).

4.2 Enhancements of the HLM Migration Mechanism

The following subchapters describe enhancements of the HLM migration mechanisms that can be implemented in the context of the constraints implied by the objectives of the mechanism. These enhancements include optimizations of the performance of the migration mechanism, the integration of additional standard interfaces, the migration of non-standard interfaces and the support of distributed identity in the context of the HLM migration mechanism.

4.2.1 Consecutive Migrations

The prototypical implementation of the HLM migration mechanism treats every migration separately and starts all over again with each new migration request. As a consequence several consecutive migrations from one environment to the same destination environment are not handled very efficiently. The set of supported interface definitions is transferred anew each time and the determination of the interface set starts all over again, despite the fact that the information has already been gathered at least partially by the previous migration.

The stateless nature of the HLM migration mechanism results from the objective that only minimal knowledge about the destination environment should be required prior to a migration. The migration does not rely on assumptions about the availability of interfaces within the destination environment. An undue slowdown of consecutive migrations does not have to be an inevitable result of this objective as simple but effective optimizations of the prototypical implementation can be implemented.

A caching mechanism can speed up consecutive migrations significantly. The sets of supported interfaces of environments that have been the destination of recent migrations can be cached at

the source and reused in subsequent migrations. This cache can be called *source cache* and can be managed with a “least recently used” strategy that deletes the oldest entries first.

When a new migration to the destination of a recent migration is initiated the source cache can provide the set of supported interfaces which does not need to be transferred anew. However, the destination environment may have gathered additional interfaces in the meantime through for example migrations from other sources or interfaces may have been added manually.

The set of supported interfaces need to be verified despite the use of the source cache. This can be done in two ways. The source can send the names of the supported interfaces it has retrieved from its cache to the destination which matches them internally and sends the additional interfaces back to the source.

Alternatively the destination environment can cache the set of supported interfaces it has sent to each source environment, in a so called *destination cache,* and can determine on its own the additional interfaces that have to be sent to each particular source environment that attempts a new migration. Figure 4.b shows the use of interface caches at both the source and destination for consecutive migrations.

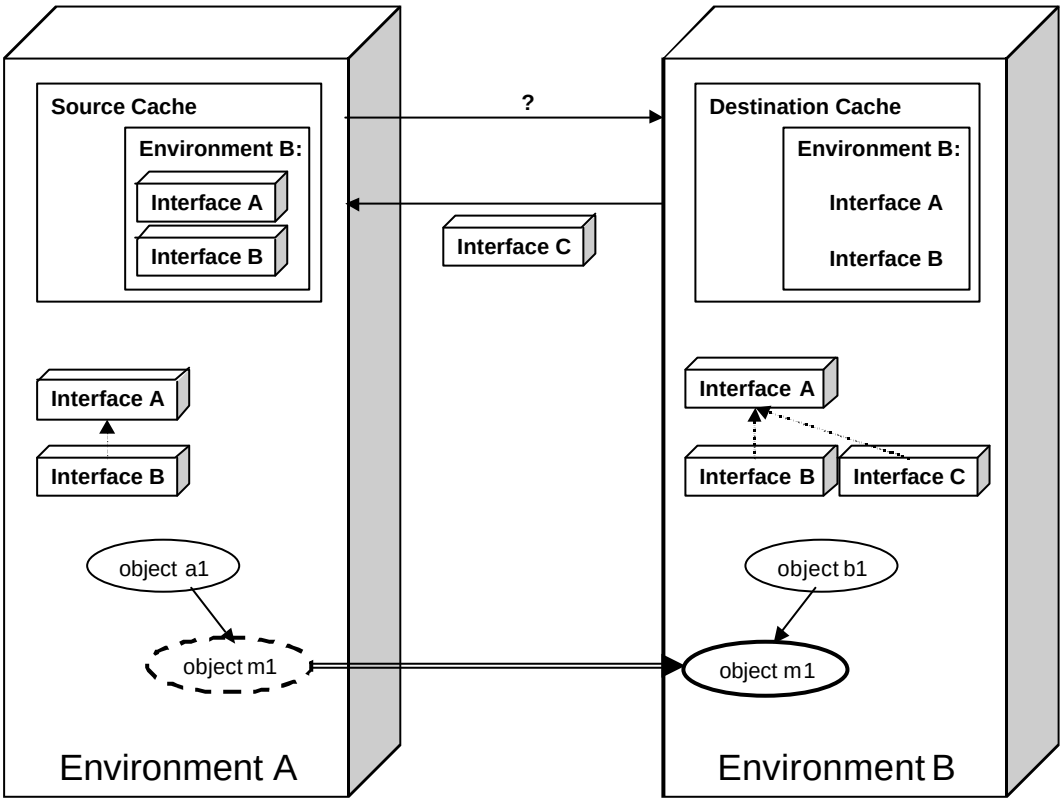

Figure 4.b: Interface caches (shown here as rectangles) can be used at both source and destination in order to streamline consecutive migrations. In the above example the source environment A reuses object definitions (shown as blocks) known from a previous migration during the subsequent migration of object m1. The set of supported interface of environment B is retrieved from the source cache and only the changes to that set are requested. The destination environment B has stored in the destination cache what interfaces have been sent to which source environment. Upon request from environment A the destination retrieves the cached information and sends only those interfaces to the source that have not already been transferred but have been defined meanwhile, in this example interface C.

In both cases a request of the source for potentially new interfaces of the destination is only necessary if the dependent interfaces are not extensible with the cached set of supported interfaces. If interfaces can also be deleted within the destination environment the request from

the source has to be sent in any case and the deletion of interfaces does also have to be reported to the source.

As several migrations between the same environments may occur back and forth both environments effectively cache the set of common interfaces that are available at both source and destination. This set will then no longer be transferred during the migration process if no changes have been made. Both the communication overhead as well as the total time of the migration process can be reduced significantly.

The basic caching mechanism optimizes only the management of the interface declaration of the supported interfaces between the participating environments and is not affected by other enhancements to the HLM migration mechanism. This optimization will also work with extensions to the migration mechanism that for example change the negotiation process as long as the destination environment is only required to provide the available interfaces (see also page 177). The optimization will probably not work with extensions to the migration mechanism that change the role of the destination environment.

4.2.2 Additional Standard Interfaces

The addition of new standard interfaces can be regarded as the "default way" to enhance the HLM migration mechanism. It may sound simple to add more standard interfaces but that task in itself can be quite complex. Additional standard interfaces have to be general enough to be working on all supported platforms and specific enough to add significant value.

For example several well known data structures can be added as standard interfaces, like lists, stacks and various forms of trees. Standard interfaces have to be specified as GOAL interface declarations and can be implemented within all supported language environments in two ways: through wrapper code or as native implementations

For some languages the implementation of a standard interface may be done through wrapper code around an existing object definition which matches the functionality of the existing object definition with the functionality defined by the standard interface. Other language environments will have to extend existing object definitions or implement the functionality from scratch.

The HLM migration mechanism will treat additional standard interfaces not differently from other interfaces. Since standard interfaces are supposed to be available within every participating environment there is no need to transfer their interface definitions during the negotiation phase. If corresponding objects of the new standard interfaces are supposed to be transferred during object migration, an appropriate ORL representation has to be implemented.

Objects of standard interfaces that obey the `OM_Owner` or `OM_Migrateable` interface will conform with the migration architecture and will be migrateable themselves using their own ORL representation. Designers of migrateable objects are free to use additional standard interfaces as long as they generate the ORL representation of their objects accordingly.

With additional standard interfaces more functionality is available to the designer of migrateable objects. Due to the added functionality the need to invent new objects may be lowered and therefore probably less interfaces have to be transferred during migration. As a side effect the added functionality may also increase the usefulness of the application specific migrateable objects within the destination environments.

Arrays

One notable exclusion from the list of standard interfaces of the HLM migration mechanism are arrays of objects. While the basic definition of arrays can be easily implemented with a syntactical extension of the GOAL language, the cross-platform aspects are more complex as almost each existing language environment implements arrays in its own specialized way.

Most object based language environments, especially hybrid ones, implement arrays via native language constructs for their creation and access. Arrays have to be declared at compile time, sometimes with a fixed size that is used to preallocate the necessary memory. Such an implementation of arrays can not be used in the context of migration as the corresponding data-structures can not be recreated during runtime within the destination environment.

A standard interface for arrays will have to allow dynamic allocation of arrays at runtime using sizes specified through variables. Languages that support dynamic allocation through their built-in constructs may implement such a standard interface for arrays through wrapper code that restricts or extends the existing functionality accordingly. The language specific syntax will have to be translated into the GOAL representation and vice versa for application specific migrateable objects that use a potential standard interface for arrays.

For all other languages that do not support the dynamic allocation of arrays a native implementation of the standard interface for arrays will have to be implemented using dynamic allocation of other native data-structures. Unfortunately, performance tradeoffs are probably inevitable as dynamic allocation of, e.g. list structures can not be as efficient as native arrays.

Conflict Management

Like arrays, some interesting candidates for standard interfaces will require more than simple interface definitions and wrapper code because fundamental characteristics of the participating language environments are addressed. The requirement for cross platform compatibility may in some cases even prohibit the definition of standard interfaces for features that are common to some language environments but foreign to others.

In order to support migration not only between as many environments as possible but also in the most functional way, the set of standard interfaces may be enlarged with several additional subsets that are only available within a limited number of languages. Such standard interfaces can be called *partially supported standard interfaces*. Likewise, the "normal" standard interfaces may be called *universally supported standard interfaces* or *universal standard interfaces* instead. Figure 4.c shows the partitioning effect of partially supported standard interfaces.

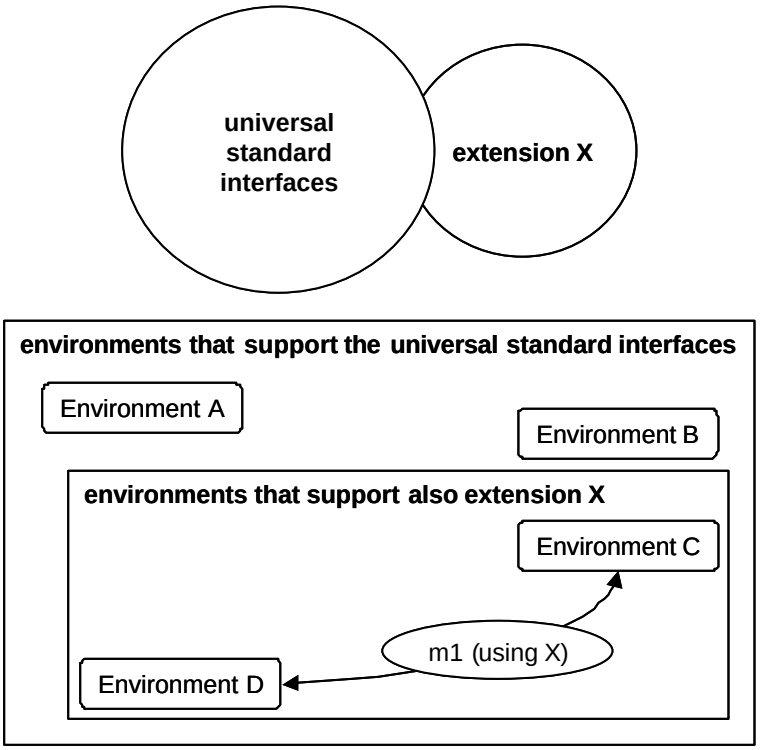

Figure 4.c: The addition of standard interfaces that are not supported by all environments will result in a partitioning of migrateability. In the above example the extension X to the set of universal standard interfaces (sets of standard interfaces are show here as circles) is only supported by the environment C and D (shown as rounded rectangles). A migration of objects that use the interfaces of extension X like object m2 (show as an ellipses) will only be possible between Environment D and E. Object that do not use extension X can be migrated freely between all environments shown.

Migration of objects that use partially supported standard interfaces will only be possible between environments with appropriate support. Attempts to migrate to destinations that offer no adequate support will fail during the implementation phase. The partially supported standard interfaces will be identified as dependent interfaces but no GOAL interface definitions can be sent to the destination as standard interfaces are implemented natively.

Whether the benefits of partially supported standard interfaces, i.e. the added functionality that can be used for some migrations will outweigh the loss of general migrateability remains an open question. The answer to this question is less a research concern but rather a matter of discussion for standardization organizations that would have to define the universal standard interfaces as well as partially supported standard interfaces if a migration mechanism such as the HLM migration mechanism is ever to be used in real applications.

4.2.3 Migration by Encapsulation

The HLM migration mechanism limits the kind of objects that can be transferred to a destination environment to objects that inherit from the `OM_Migrateable` interface either directly or indirectly. The applicability of the HLM migration mechanism can be enlarged significantly if this requirement for objects to be transferred is lowered.

Support for the migration of objects that do not inherit from the `OM_Migrateable` interface can be added to the HLM migration mechanism as an enhancement. Objects that are not migrateable themselves can be encapsulated by `OM_Migrateable` objects. An `OM_Migrateable` object can be designed to include the objects it encapsulates as part of its own representation.

An object is a so called *encapsulated object* if the following conditions hold:

1. the object is referenced only by a single `OM_Migrateable` object, called its *anchor object*,
2. the object does not reference other objects themselves except:
   a. `OM_Migrateable` objects that are included in the migration set,
   b. other encapsulateable objects of the same anchor object, or
   c. singular objects, i.e. objects of standard interfaces that are represented as atomic values

If these restrictions apply, the anchor object can provide the necessary functionality to check, represent, negotiate, rebuild, reinitialize and deactivate/activate or reactivate/abort the encapsulated objects it references. An anchor object will include the objects it encapsulates as part of its own ORL representation and will essentially handle the whole migration process for the encapsulated objects.

An encapsulated object can only be referenced by a single anchor object as a migration would otherwise have to be coordinated among several anchor objects in the same way as between `OM_Migrateable` objects and `OM_Owner` objects. Encapsulated objects are supposed to lower the requirements of the HLM migration mechanism not the contrary.

Encapsulated objects cannot be referenced from other objects because these may be left with dangling references after the encapsulated objects are migrated. The designer of the respective application is responsible to prevent such situations. Appropriate development tools can help to detect whether references to encapsulated objects that are created by anchor objects are passed to other objects (see also chapter 3 page 121).

Encapsulated objects are free to reference other encapsulateable objects as long as these can be handled recursively by the same anchor object. Encapsulated objects can reference other `OM_Migrateable` objects of the migration set. These references can be represented by the anchor objects via the corresponding object migration identifiers and can also be reestablished within the destination environment.

Encapsulated objects can also reference other objects that are not encapsulated as long as they do not pass references to the other objects. A condition that can also be detected by appropriate development tools. As the anchor object does only handle the migration for

encapsulated objects the non-encapsulated objects referenced by encapsulated objects can not be migrated. Figure 4.d illustrates the requirements for encapsulated objects.

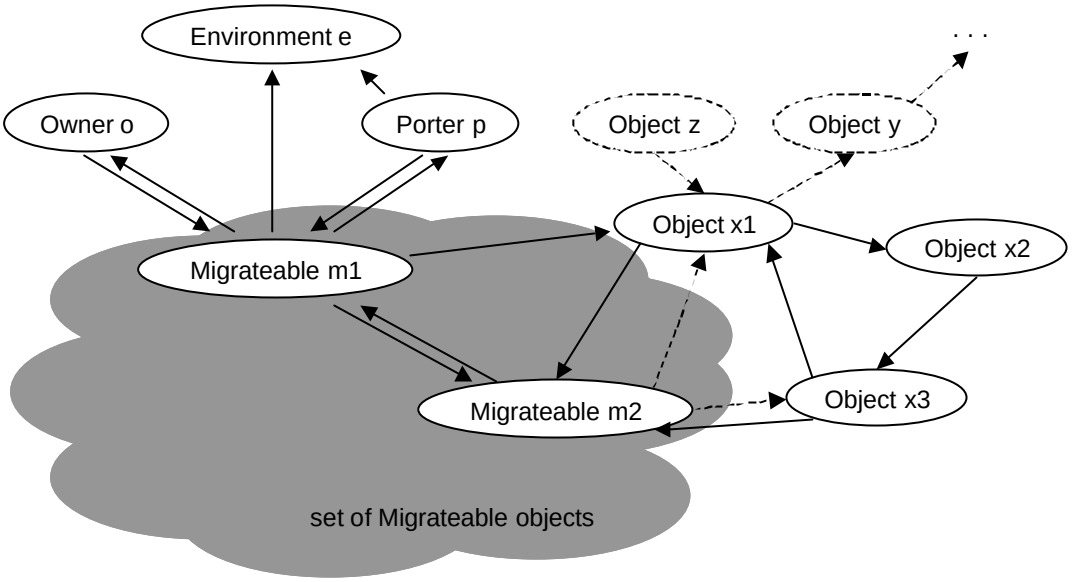

Figure 4.d: Encapsulated objects can be referenced only by a single `OM_Migrateable` object either directly or indirectly. In the above example the objects x1, x2 and x3 are encapsulated by anchor object m1. References from encapsulated objects to other `OM_Migrateable` object of the migration set (e.g. x1 to m2) or to other encapsulated objects of the same anchor object (e.g. x3 to x1) are permissible. References to other objects (e.g. x1 to y) or from other objects (e.g. z to x1) are prohibited (references that are not permissible are shown here as dashed arrows).

A good example of an encapsulated objects is a datastructure like a list that is used by an `OM_Migrateable` objects to manage other `OM_Migrateable` objects. The objects the list consist of do not have to be transferred as the list can also be reconstructed within the destination environment. The anchor object of the list has to include the object migration identifiers of the elements the list refers to as part of its ORL representation and has to reconstruct the list accordingly as part of the reinitialization step of the transfer of state phase.

The encapsulated objects can only use standard interfaces and interfaces that are descendants of the migration architecture. Otherwise the migration of encapsulated objects will not succeed as the necessary interfaces are probably not supported by destination environments and can also not be added through extension.

An actual migration of encapsulated objects will be processed in the usual way as the anchor object performs all steps of the migration algorithm on behalf of the encapsulated objects. The anchor objects are the only objects referencing the encapsulated objects and therefore the only objects able to send messages to them.

The anchor objects therefore “knows” whether methods have been invoked for the encapsulated objects and is able to maintain an invocation counter for each encapsulated object it references. In order to maintain control over the encapsulated objects, the anchor object is not supposed to pass references to the encapsulated objects to other objects (see chapter 3 page 99).

Whether the interfaces of the encapsulated objects have to be transferred as part of the interface set will be determined by the negotiation algorithm in the normal way. For the transfer of state the encapsulated objects will be represented by their anchor object as part of its ORL representation and not as separate entities with their own object migration identifiers.

As a consequence the encapsulated objects will not be recreated by the `OM_Porter` object of the destination environment but as part of the `initialize_after_migration()` method of

their anchor object. Similarly the encapsulated objects will not have to provide their own deactivate/release, activate or reactivate methods but the necessary work will have to be performed by the corresponding methods of their anchor object.

The whole process can be called *migration by encapsulation* and typical candidates for this kind of migration are all kinds of supportive datastructures like lists, stacks and trees. These versatile objects are used in all kinds of applications but their use in application of the HLM migration mechanism would require special `OM_Migrateable` based implementations for each of them.

Migration by encapsulation is able to widen the interface set of migrations and therefore the applicability of the HLM migration mechanism. The design of applications that use migration can be simplified as fewer objects that inherit from the `OM_Migrateable` interface need to be defined. Most importantly the interfaces of the encapsulated objects do not need to be changed but can be used as they are.

4.2.4 Partial and Incremental Migration

The HLM migration mechanism does not support distributed identity due to its focus on existing environments most of which do not support any kind of *location independence*. The prototypical implementation of the migration mechanism always performs a so called *complete migration* as no relationships remain between the objects of the source and the transferred objects.

The addition of support for distributed identity to the migration mechanism can on the other hand open up new ways of performing object migrations. With distributed identity, related objects do not have to be migrated along with the object the migration has been initiated for. Relationships among objects can be extended in the context of distribution between the participating environments.

A migration that that does not transfer objects completely can be called a *partial migration* as the relationships between the root object and its related objects at the source still exists after migration. Furthermore related objects can be transferred subsequently as the need arises by what can be called *incremental migrations*.

The techniques that are used to implement location independence differ largely between distributed systems. With regard to object migration three forms of location independence can be distinguished: non-transparent remote references, transparent remote references as well as the use of proxy objects.

Non-transparent Remote References

If location independence is implemented through non-transparent remote references that are built into language environments a developer of a distributed application will have to differentiate between local references and remote references as well as between local invocations and remote invocations. As a consequence the distributed identity of objects will differ from the local identity and has to be expressed within the program code explicitly.

Only few systems differentiate between local and remote references like Amber [CA+1989b]. The Common Object Model (COM) defined by the Microsoft Corporation also uses so-called Monikers to identify distributed resources. The COM environment is available for a number of object-based languages and implements transparent remote procedure calls for COM objects.

In the context of non-transparent remote references objects of distributed applications have to be designed with conditional behavior that depends on local and remote operations. Heterogeneous object migration can only be performed if both source and destination environments support compatible non-transparent remote references.

In order to perform migrations in the context of non-transparent remote references local references of objects of the source environment to the object to be migrated have to be substituted with remote references to the migrated object after migration. If the objects to be migrated reference objects of the destination remotely prior to migration the reverse operation has to be applied to these relationships during migration.

References of the object to be migrated to objects of other environments are left unchanged during migration. References of objects of other environments to the migrated object have to be

updated with the new location information. Remote method invocations have to be used between the migrated object and its related objects that remain at the source or elsewhere.

Migrations between environments that use non-transparent remote references and environments that use other forms of location independence can not be supported without changes to the definitions of the objects to be migrated and changes to the participating environments. The transfer of computations of objects is not possible as local references from activation records can usually not be exchanged for non-transparent remote references if the referenced objects are migrated (see also page 179).

Despite the fact that the HLM migration mechanism can probably be enhanced to perform object migrations in the context of non-transparent remote references for homogeneous environments support for this kind of location independence is not investigated further. Apart from an inevitable increase of complexity in the context of migration, non-transparent remote references are only seldom used as newer techniques have been developed.

Transparent Remote References

Transparent location independence is achieved when remote references are built into language environments that do not have to be distinguished from local references and remote method invocations can be programmed like local method invocations. Every reference to an object will be a remote reference and the code of applications does not have to differentiate whether methods of local or remote objects are invoked. In some cases even references of activation records can be transparent remote references as well.

Only few systems as for example Emerald [Jul1993] implement transparent remote references in the form described above and notably Emerald is deliberately designed to implement transparent object migration. Interestingly enough Emerald transfers also the activation records of methods that have been invoked for objects (see also page 179).

In the context of transparent remote references objects can be migrated freely among environments. If activation records are also implemented using transparent remote references computations of objects can be migrated freely as well. The remote references of related objects or activation records only have to be updated with the new location information of the migrated objects (see also chapter 2 page 42).

Unfortunately, heterogeneous migrations will only be possible if both the source and destination environments implement transparent remote references in a compatible way. Interoperability with other forms of location independence will not be possible unless changes to the definitions of the migrated objects are applied as well as to the participating environments.

The HLM migration mechanism can probably be enhanced to migrate arbitrary sets of objects in the context of transparent location independence. An object that references an object to be migrated will not notice when that object actually migrates to another environment. The migration of objects will be transparent. If references of activation records are also remote references even objects that are referenced as parameters in computations can be migrated. If a method has been invoked for an object that object can not be migrated unless the activation records can be migrated as well (see also page 179).

Despite the obvious advantages of transparent remote references their support in the context of the HLM migration mechanism is not investigated any further. Transparent remote references are only used by very few systems and their support in a heterogeneous context of would require significant changes to the participating environments.

Location Independence through Proxy Objects

In the context of object-based environments, location independence can be implemented in the form of proxy objects that are established as local representations of remote objects. Proxy objects encapsulate remote references and implement remote method invocation through the forwarding of local messages (see also chapter 2 page 42).

A proxy object participates in the local message passing but relays messages through a network protocol to its corresponding remote object, which is called here the *sovereign object* of

the proxy object. The sovereign object executes the appropriate method and sends the result back to the proxy that returns it to the message sender. Whether the message is executed locally or remotely is indistinguishable for the original sender of the message apart from a possible delay due to the communication overhead.

Proxy objects are frequently used within distributed systems to achieve location independence of system objects as for example in SOS [SG+1998] or language objects as for example in Brouhaha [DNX1992] or DOWL [Ach1993b]. Agents system like Voyager [Gla1998] also employ proxy objects for the communication between agents.

Proxy objects can be employed in the context of migration to maintain relationships between objects. A reference to an object that is migrated can be maintained through the creation of a proxy object in place of the migrated object[23]. Altogether six different cases have to be distinguished for the management of proxy objects during migration:

1. When an object is migrated to another environment a proxy object will have to be created in its place in order to let related objects access the migrated object transparently.
2. References of the object to be migrated to other objects of the source have to reestablished once the object is migrated through the creation of proxies within the destination environment.
3. Proxy objects within the source environment that are used by the object to be migrated as references to objects of the destination environment have to be destroyed during migration and local references can be established within the destination environment once the object is migrated. If the proxies of the source environment are used by other objects of the source the proxies have to be maintained though.
4. Proxies of the object to be migrated that exist within the destination environment have to be replaced by local references once the object is migrated.
5. Proxy objects of the source that are used by the object to be migrated to reference objects of other environments have to be reestablished within the destination environment and destroyed at the source unless they are used by other objects of the source environment.
6. Proxies objects of the object to be migrated within other environments need to be updated with the new location information after migration.

Although confusing at first sight, the interplay of migration and proxy objects is actually quite simple. During migration all relationships to and from the migrated objects will be turned from local references to proxy-based references and vice versa within both the source and destination environments. Proxy-based references to and from other environments will be migrated as well or updated with the new location respectively. Figure 4.e illustrates migration in the context of proxy objects.

Proxy objects forward local message to their sovereign objects. To do so proxy objects marshal a message passed to them including its parameters and send the serialized message over the network to their remote object. The sovereign object has to demarshal the message, invoke the corresponding method and send the result back in the same way. The proxy will again demarshal the result and pass it back to the local sender of the original message.

In the context of object migration potentially every object has to be able to work with its own proxies that can be created in other environments. This condition is obviously prohibitive as it would require proxy implementations for all objects of an application or at least for all objects that can potentially be referenced by the objects that can be migrated.

A second problem with the management of proxies in the context of migration is the replacement of the migrated object with its own proxy within the source environment. A seamless replacement can only be achieved if the particular environment provides a built-in operation to replace an object with its proxy while keeping all relevant references consistent.

[23] There are several ways to implement proxy objects, but only the most common one that uses proxy objects for all references between environments will be discussed here.

If for example memory addresses are used as local references this operation would have to replace the object at the exact same memory location and all other implementational details like method-tables would also have to match perfectly. This condition is also obviously unrealistic as none of the existing language environments provides such an operation.

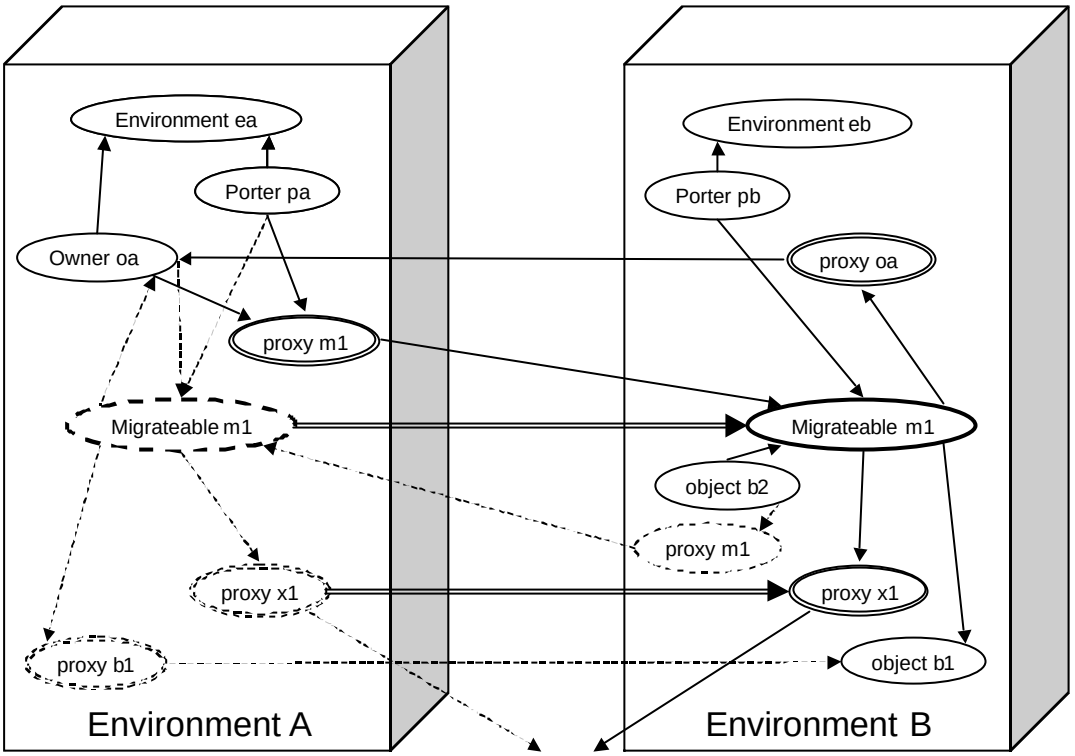

Figure 4.e: Partial migration can be implemented in the context of proxy objects. In the above example the Migrateable m1 is transferred to environment B and the proxy object m1 is created in its place to be referenced by Owner oa (case 1 as mentioned above). The reference of m1 to oa is maintained through the creation of proxy oa (case 2). Proxy b1 that was used by m1 to reference object b1 is replaced by a local reference in environment B and deleted in environment A unless its is still used (case 3). Proxy m1 that was used within environment B by object b2 to reference m1 is replaced by a direct reference from b2 to the now migrated m1 (case 4). Proxy x1 that was used by m1 to reference an object of another environment is moved to environment B or copied if it is still used within environment A (case 5). The update of proxies of m1 in other environments is not shown (case 6) (Proxies that are created or that may be deleted during migration are shown here with double lines and dashed double lines respectively).

As a consequence the object to be migrated and its related object will have to cooperate to perform the necessary handling of proxy objects. The object to be migrated has to be able to inform all objects that reference it about its replacement with a proxy object. This consideration is just a reiteration of the one that led to the development of the migration architecture of the HLM migration mechanism (see also chapter 3 page 75).

The HLM migration mechanism can be enhanced to support distributed identity through proxy objects even in the context of heterogeneous systems. Proxy objects can be defined as an addition to the migration architecture and the migration algorithm can be augmented to support proxy objects during migration.

Migration Architecture

A proxy object must be able to forward all messages defined for the object they stand for. The object that is represented by a proxy object must be able to handle all forwarded messages in order to become what can be called a *sovereign object* to all its proxy objects. Proxy objects

can be defined through interfaces that inherit from the interface of the objects they stand for[24]. The interface of a proxy object will be called *proxy interface*. The interfaces of objects that are augmented to become sovereign objects can be called *sovereign interfaces*.

To allow the use of proxy objects in the context of the HLM migration mechanism all `OM_Migrateable` objects as well as all `OM_Owner` objects need to become sovereign objects. For each sovereign interface a proxy interfaces need to be defined, which as a convention can be named with a `_Proxy` suffix to the name of the sovereign interface.

Fortunately the generation of proxy interfaces does not interfere with single inheritance as additional descendants can be defined for sovereign objects. The descendant will have separate proxy objects though as they may redefine or add signatures. Figure 4.f shows the necessary enlargements of the migration architecture.

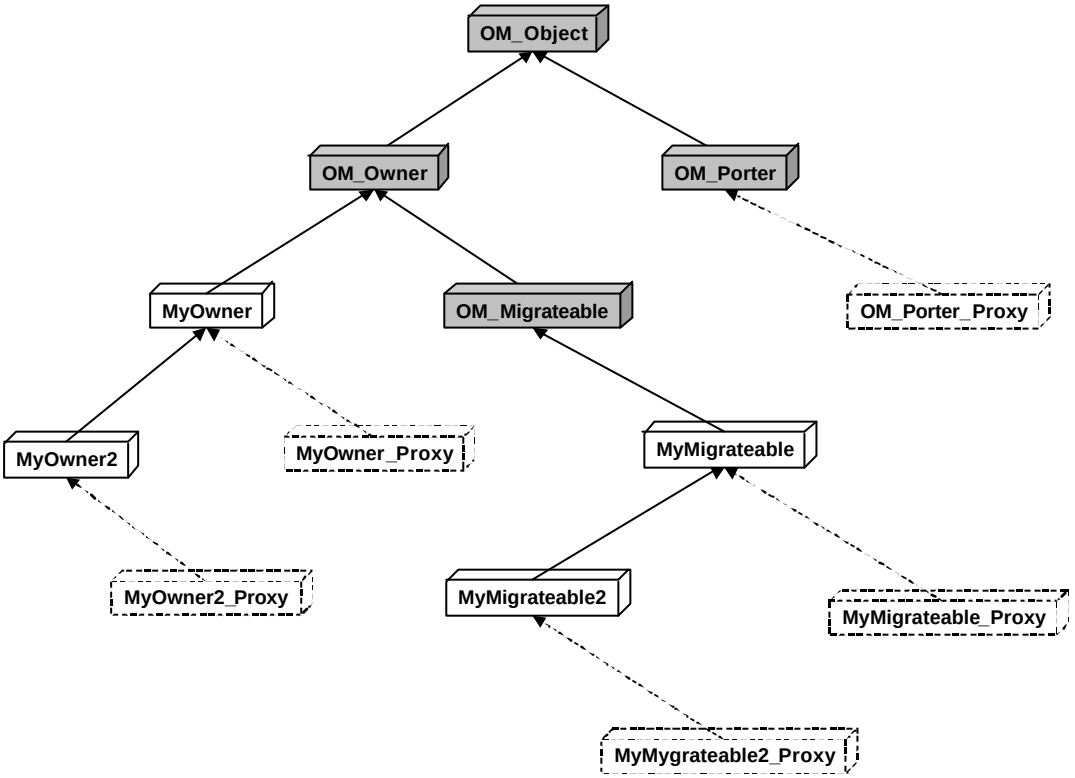

Figure 4.f: The migration architecture in the context of proxy objects. Each interface that inherits from the `OM_Owner` or `OM_Migrateable` interfaces is augmented to become a sovereign interface and a proxy interface is generated as a descendant.

Both the proxy interfaces as well as the sovereign interfaces have to contain the necessary code for the serialization of message parameters which has to be based on the standard interfaces of the HLM migration mechanism. The necessary serialization code can be generated statically at development time by appropriate development tools (see also chapter 3 page 121).

Migration Algorithm

Several phases of the migration algorithm of the HLM migration mechanism have to be augmented in order to support proxy objects. Significant changes have to be applied to the check phase, the transfer of state phase and the completion phase. The initiation phase can be simplified, and the negotiation and transfer of semantics phases do not need to be changed at all. Figure 4.g shows an example that is used to illustrate the support of proxy objects.

[24] Other approaches to define proxy objects, e.g. as separate interface with the same signatures will not work in context of the HLM migration mechanism due to the negotiation phase that is necessary to overcome heterogeneity.

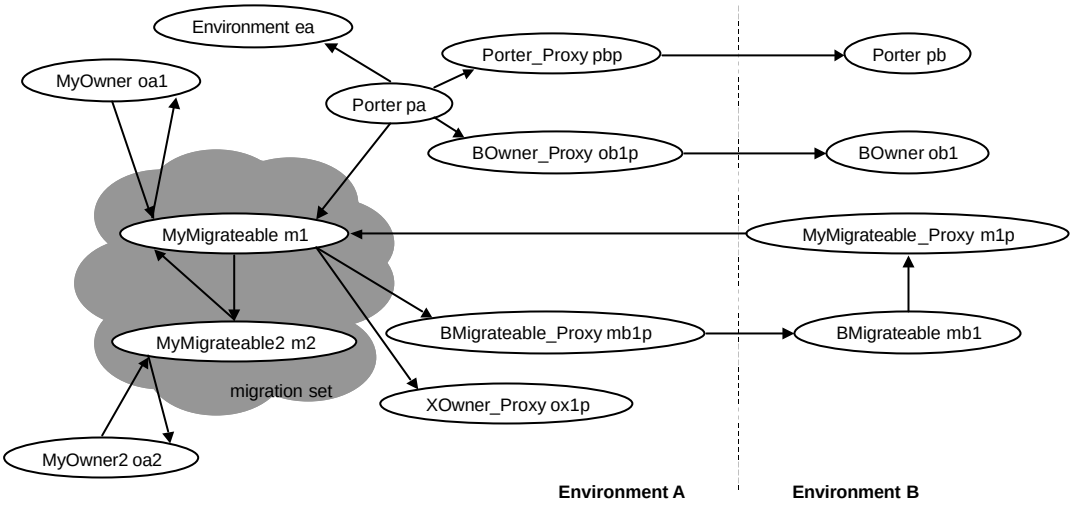

Figure 4.g: The above example is used to illustrate the changes that have to be applied to the migration algorithm of the HLM migration mechanism in order to support proxy objects. The `MyMigrateable` object m1 is supposed to be migrated to environment B to the `BOwner` ob1 that already references m1 via its proxy m1p. The `BMigrateable` mb1 of Environment B is referenced by m1 via proxy mb1p and m1 also references the proxy ox1p of an `XOwner` ox1 of another environment that is not shown here for reasons of space. The `MyMigrateable` m1 is also related to the `MyOwner` oa1 and the `MyMigrateable2` m2 which itself is related to `MyOwner2` oa2. The only case that is not shown here is a proxy object of m1 or m2 that may exists within a different environment.

In the context of location independence through proxy objects a migration request can contain a reference to a remote `OM_Porter` object that is represented within the source environment through a proxy object. The communication between the `OM_Porter` objects of the source and the destination can be implemented as a number of remote method invocations. Additionally an `OM_Owner` object of the destination can be defined as a target for the migration as well.

Initiation

If a proxy object is used to reference the `OM_Porter` object of the destination within the migration request the initiation phase of the migration algorithm can be simplified. The attempt to establish a network connection can be replaced by a simple remote method invocation for the `OM_Porter` object of the destination environment.

Check

The check phase does no longer have to perform a migration check that determines whether the dependent objects of the root object of the migration request can be migrated. Rather each related `OM_Migrateable` object of the root object can decide independently whether it is part of the migration or not. The relationships of the root objects to other objects will not have to be cut but will have to be replaced by proxy objects either at the source or at the destination.

The migration check will still be a recursive descent that starts from the root object of the migration request. Every related object that inherits from the `OM_Migrateable` interface is asked whether it wants to be part of the migration. If so, the object is added to the migration set, its interface is added to the set of interfaces to process, i.e. the *negotiation set* and the recursive decent continues for its components.

If the related object does not want to part of the migration, the proxy interface of the object is added to the negotiation set and the recursive descent backtracks. If the related object is represented by proxy object, the proxy interface is added to the negotiation set and the recursive descent backtracks. These steps are necessary as related objects that are not

migrated will have to be referenced via proxies from the destination and proxies that reference objects of other environments have to be recreated within the destination environment.

Despite the use of proxy objects, computations of objects will still not be migrateable even with an enhanced HLM migration mechanism as references of activation records can not be redirected from objects to proxies and vice versa without changes to the existing language environments (see also page 179). The objects to be migrated therefore still need to be inactive.

Fortunately activation records of remote environments will reference the objects to be migrated via proxy objects and will not be affected by migration. Only the location information of the corresponding proxy objects have to be updated after the migration. Therefore no additional inactive checks to the local ones already described are necessary (see chapter 3 page 99).

The migration set that is constructed for the example of Figure 4.g may contain the objects `m1` and `m2`. As a consequence, the negotiation set will contain the interfaces `MyMigrateable`, `MyMigrateable2`, `MyOwner_Proxy`, `MyOwner2_Proxy`, `BMigrateable_Proxy`, and `XOwner_Proxy`.

Negotiation

The negotiation phase will be performed in exactly the same way as before. The supported interfaces of the destination environment will be requested and the set of interfaces received which will now also include the proxy interfaces available within the destination. The negotiation algorithm will determine the interface set which will now also include proxy interfaces for all proxies that have to be established anew within the destination environment.

In the context of the example of figure 4.g the negotiation process will probably determine that the interface `BMigrateable_Proxy` is already supported by environment B. Under the assumption that none of the interfaces of the application of environment A is supported by environment B the interface set will at least contain the interfaces `MyMigrateable`, `MyMigrateable2`, `MyOwner_Proxy`, `MyOwner2_Proxy`, and `XOwner_Proxy`, as well as `MyOwner` and `MyOwner2`.

Transfer of Semantics

The transfer of semantics phase will also be performed in the same way as before. Interface definitions will be generated within the source environment for all interfaces of the interface set. These interface definitions will be sent to the destination environment and compiled there into native object implementations which will be loaded dynamically.

Transfer of State

The transfer of state phase of the HLM migration mechanism starts with the representation of the objects to be migrated, deactivates the objects of the migration set, transfers the representation, and rebuilds and reinitializes the objects within the destination environment. The order of events will be the same as before but the individual operations have to be modified in order to support proxy objects.

- Representation

The ORL language that is used for the representation and transfer of the objects of the migration set has to be changed in order to support proxy objects. The relationships between the objects to be migrated and their related objects have to be represented as proxy objects in addition to the usual representation of the state of the objects in the migration set.

The ORL representation of proxy objects has to contain enough information that the relationship between proxy objects and their sovereign objects can be determined and that messages can be forwarded to the sovereign objects from their proxies. The ORL representation therefore has to include information about the location of objects or their environments respectively as well as information that enables the identification of particular objects within their environments.

In the context of the HLM migration mechanism the simplest form of location and identification information can be used as more sophisticated techniques can be added in future versions of the mechanism. The location information to be added to the ORL representation can consists of

the TCP/IP address of the environment of the particular object as well as of an integer number called *object identifier* that identifies the particular object within its environment. As the textual representation of the TCP/IP address the textual dot "`.`" delimited four byte format can be used which has to be prefixed by a separator for example an ampersand "`&`" sign.

The object identifier of an object can be defined by the `OM_Porter` object of an environment using a request that included a reference to the object. The `OM_Porter` generates a new object identifier and registers the object using the object identifier within directory of all objects of the corresponding environment that can be access via proxy objects. The `OM_Porter` will also maintain a so called *proxy directory* of all proxy objects of its environment.

When the objects of the migration set are migrated the related objects of the source environment have to be addressed via proxy object from the destination and vice versa. The proxy objects of related objects that reside within different environments have to be recreated within the destination environment as well. The ORL representation has to be changed to allow the management of proxies to be performed before the relationships of the migrated objects are reestablished within the destination environment.

The ORL representation is constructed as before with two iterations through the migration set although the operations performed and the format of the ORL representation differ. The first iteration assigns an object migration identifier to each object within the migration set as well as to each related object. It will be sufficient that a proxy objects store the object migration identifier on behalf of its sovereign object in order to minimize the communication overhead.

For each object of the migration set the object migration identifier, the interface of the object, the TCP/IP address of its environment as well as the object identifier are added to the ORL representation and an iteration of the related objects is performed. If an object does not have an object identifier it will have to register with the `OM_Porter` object of its environment as an object that can be addressed by proxy objects and will receive its object identifier in return.

For each related object that is not a member of the migration set the object migration identifier, the corresponding proxy interface of the object, the TCP/IP address as well as the object identifier of the sovereign object are added to a separate ORL representation that can be called the *proxy representation*.

For each related object that is represented by a proxy object within the source environment, the object migration identifier, the proxy interface, and the TCP/IP address and object identifier of the sovereign object are added to the proxy representation. Related objects that are members of the migration set will be processed as part of the iteration.

After the first iteration has been performed the first part of the ORL representations has been constructed as a combination of the individual representations of the objects of the migration set. The proxy representation that has been gathered along the way is added as a new second part to the ORL representation; delimited by a pair of bar "`|`" signs.

The proxy representation contains the information about the relationships of the objects of the migration set that have to be reestablished within the destination environment in the form of proxy objects. The proxy representation is not a one-to-one representation of the corresponding related objects within the source environment but rather a representation of how these objects are going to be references from the destination environment, i.e. via proxy objects.

The second iteration through the migration set adds an individual representation of each objects state to the ORL representation. For each object of the migration set the object migration identifier, the textual representations of its components that are singular objects as well as the object migration identifiers of all related objects each preceded by a colon "`:`" sign are added.

The related objects do not need to be represented as they will be references via newly created proxy objects from the destination environment. Only the state of those objects that are actually migrated needs to be represented. Excerpt 4.a shows the ORL representation that is generated for the example of Figure 4.g.

```
1 MyMigrateable &127.0.0.1 5362
5 MyMigrateable2 &127.0.0.1 5463
|
2 MyOwner_Proxy &127.0.0.1 3645
3 BMigrateable_Proxy &127.0.0.10 5362
4 XOwner_Proxy &127.0.0.99 8293
6 MyOwner2_Proxy &127.0.0.10 3847
|
1 (:2 :3 :4 :5 )
5 (:1 :6 )
```

Excerpt 4.a: The ORL representation for the example of figure 4.g. The TCP/IP addresses and object identifiers that are used to distinguish environments and objects are arbitrary.

- Deactivation

After the ORL representation has been constructed the deactivation of the related objects is performed with another iteration through the migration set. The `OM_Porter` sends a `deactivate()` message to the objects of the migration set. Each object sends its owners and components that are not members of the migration set and that are not proxy objects a `deactivate_component()` or `deactivate_owner()` message respectively.

The related objects that reside within the source environment will be deactivated, i.e. these objects will be informed that their relationship to the particular member of the migration set is not longer active. Related objects that are members of the migration set will be processed in the same way as part of the iteration.

There is no need to deactivate proxy objects as the corresponding remote owners or components communicate with the objects of the migration set through proxy objects within their remote environments. These location information of these remote proxies needs to be updated after the migration has been performed.

With a successful deactivation of the relationships of the objects of the migration set, the source environment will reach “ready to commit” state. I.e. source environment is ready to either destructively complete the migration or roll back any changes made so far. Figure 4.h shows the state of the migration process after deactivation has been performed.

After the successful deactivation the ORL representation will be transferred to the `OM_Porter` object of the destination environment. The `OM_Porter` object recreates the objects of the migration set as before with a rebuild and a reinitialization step. Both of these steps need to be modified to support proxy objects.

- Rebuild

The `OM_Porter` object of the destination environment parses the first part of the ORL representation, creates each represented object from the corresponding interface, and assigns it the individual object migration identifier. The newly created object is then added to the set of objects recreated for the current migration that is called the *rebuild set*. The first object of the ORL representation will also be regarded as the root object of the migration.

The recreated objects will be also assigned their previous TCP/IP address and object identifier as well as separately a newly determined object identifier within the destination environment. The objects will be added to the directory maintained by the `OM_Porter` object under their new object identifier.

The `OM_Porter` then continues for the second part of the ORL representation, creates each represented proxy object and assigns it the individual object migration identifier, if such a proxy does not already exists within the destination environment. The newly created proxy object is then added to the set of proxy objects recreated for the current migration that can be called the *proxy set*. Existing proxies that match the second part of the ORL representation will be added as well.

For each recreated proxy object whose TCP/IP address matches that of the destination environment the `OM_Porter` performs a lookup in its directory of object identifiers and retrieves the corresponding object of the destination environment. This match will only be possible for proxy objects of the source environments that act on behalf of objects of the destination environment.

As the object of the rebuild set are supposed to reference these objects directly within the destination environment the `OM_Porter` object will replace the proxy stored in the proxy set with the object retrieved from the directory and will also reassign the particular object migration identifier accordingly.

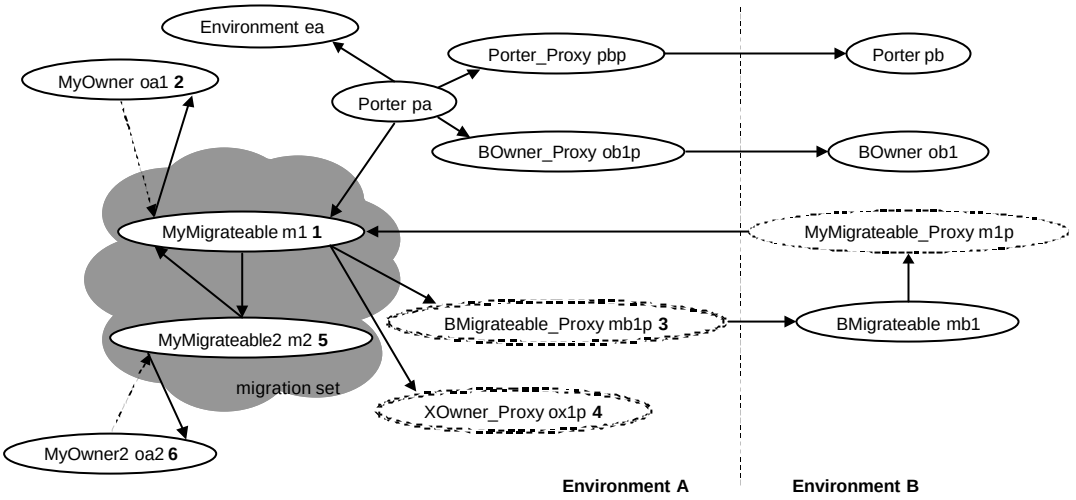

Figure 4.h: After the representation and deactivation steps of the transfer of state phase have been performed to the example of figure 4.g all references to objects of the migration set are inactive and object migration identifiers have been assigned to each related object or its proxy (references that are no longer active are shown here as dashed arrows and the object migration identifiers as bold numbers, object identifiers as well as the directory kept by the `OM_Porter` objects are not shown for reasons of space, objects that will be deleted by subsequent phases are shown with dashed lines, objects that can be deleted with double dashed lines)

In the example shown in figure 4.h the proxy mb1p of interface `BMigrateable_Proxy` will be recreated within environment B but matched and replaced with the object mb1 of interface `BMigrateable`. A new proxy ox1p of interface `XOwner_Proxy` will be created within environment B as well as the proxies oa1p of interface `MyOwner_Proxy` and oa2p of interface `MyOwner2_Proxy` all of which are not affected by the replacement operation.

- Reinitialization

After all objects have been created a `initialize_after_migration()` message is sent to the root object of the rebuild set. The root object parses its ORL representation and reestablishes its references to the other objects of the rebuild set as well as of the proxy set with the help of the `OM_Porter` that returns a reference to the particular object or proxy object for each lookup of an object migration identifier.

The root object will then send an `initialize_after_migration()` message or a similar message with additional information to its related objects that are members of the rebuild set. The related objects will process their part of the ORL representation in the same way as the root object and will send appropriate messages recursively.

After the reinitialization phase the migrated objects will have established only direct relationships to objects within the destination environment. The destination environment will therefore have reached “ready to commit” state as the migrated objects can be fully activated or all changes so far can be rolled back.

Completion

During the completion phase either a commit or an abort is performed. With a commit, the migrated objects within the destination environment are activated and the object of the migration set within the source environment are deleted. With an abort all newly created objects within the destination environment are deleted and the objects in the migration set within the source environment are reactivated. All these operations have to be augmented in order to support proxy objects.

- Activation

The activation step is performed by the `OM_Porter` of the destination environment in the case of a commit. An `activate()` message will be sent to the root object of the migration which performs additional initializations and sends an `activate()` message including a reference to itself to all its related objects that are not represented by proxy objects within the destination environment.

Each related object that is not in the rebuild set will check whether it references a proxy object of the migrated object. In order to do so, the TCP/IP address and the object identifier stored by the proxy object and the previous TCP/IP address and object identifier stored by the migrated object are compared. If a match is detected the reference of the related object to the proxy will be replaced with a reference to the migrated object. With this operation a reference to a once remote object via a proxy is replaced with a local reference to the migrated object.

Proxy objects that represent sovereign objects of other environments will perform any necessary initialization upon receipt of the `activate()` message. Each related object that is part of the rebuild set will also perform its own initialization and will send `activate()` messages recursively to its related objects including an appropriate reference.

When the activation of the migrated objects has been performed within the destination environment the corresponding objects of the source environment need to be informed of the new location of the migrated objects. The `OM_Porter` object will therefore have to collect an new ORL Representation of all objects of the rebuild set that includes the object migration identifier, TCP/IP address and the new object identifier of each object as defined by the destination environment. This ORL representation will be called *update information*.

The update information is sent back to the `OM_Porter` of the source environment as an acknowledgement that the activation was successful. The `OM_Porter` will parse the ORL representation and assign the new TCP/IP address and object identifier as a separate information to each object in the migration set.

- Release

The release step is performed by the `OM_Porter` of the source environment in the case of a commit. An `release()` message is sent to the root object of the migration set which creates a proxy object of itself. The root object sends its related objects that are not members of the migration set and not proxies themselves a `replace_component_with_proxy()` or `replace_owner_with_proxy()` message depending on the mutual relationship including a reference to itself as well as to its proxy.

The related objects will replace their reference to the migrated object with the reference to the proxy. The root object will delete its references to proxy objects and send a `release()` message recursively to its related objects that are members of the migration set. With end of the release step the migration is performed successfully and the `OM_Porter` object can delete the migration set. Figure 4.i shows the example of figure 4.g after the migration has been performed successfully.

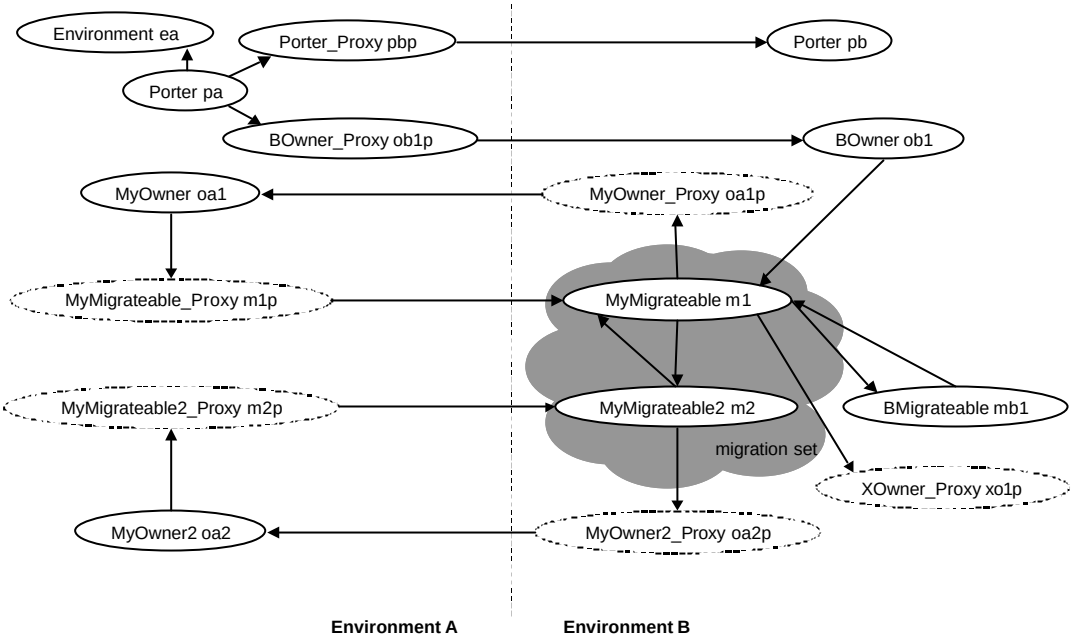

Figure 4.i: The migration has been performed successfully for the example of figure 4.g. The migrated objects have been replaced by proxy objects within the source environment and their related objects are referenced from the destination environment via proxies. Proxies of objects of other environment have been transferred and relationships with objects of the destination environment have been replaced by direct references within the destination environment.

- Abort

In the case of an abort of the migration during the completion phase all objects of the rebuild set and proxy set are deleted by the `OM_Porter` of the destination environment. The only exception are non-proxy objects of the proxy set, i.e. objects of the destination environment that have replaced proxies within the proxy set as well as proxies that existed previously.

- Reactivate

In the case of an abort of the migration during the completion phase, the `OM_Porter` of the source environment sends a `reactivate()` message to the root object of the migration request. The root object will forward the message to its related objects recursively, which will reestablish their relationships and will also forward the message recursively (see page 158).

The HLM migration mechanism can not only be extended to support proxy objects it will also be able to perform partial migrations in the context of heterogeneous systems. Since proxy object can be added to existing languages they can be regarded as the default way to implement distributed identity in the context of the HLM migration mechanism. The interoperability with other forms of location independence can always be achieved through the implementation of proxy objects in addition to for example transparent remote references.

Despite the possible use of proxy objects to support distributed identity the HLM migration mechanism will not be able to transfer computations of objects during migration as references of activation object to object to be migrated can not be replaced by references to proxy objects without changes to the participating environments (see also page 179).

Implementation of Proxy Objects

Although feasible the use of proxy objects within the HLM migration mechanism implies a number of subtle consequences for the implementation of both the HLM migration mechanism as well as the proxy objects. The problems that arise from the interdependence of the migration mechanism and the proxy objects can only be mentioned here briefly and some hints at possible solutions can be given. An exhaustive discussion of the more general questions behind these problems lies beyond the scope of this work.

The HLM migration mechanism is intended to work with only minimal assumptions about the destination environments. Although an implementation of the mechanism is required for the participating environment no assumptions about the availability of interfaces within destination environments are made.

As a consequence the implementation of proxy objects in the context of the HLM migration mechanism have to be able to be added as extensions to particular destination environments, i.e. GOAL interface definition have to be available for each proxy object. This implication essentially prohibits the use of existing proxy implementations in the context of the HLM migration mechanism.

Existing implementations of location independence as for example CORBA [Sie1996] generate proxy implementations at development time for specific server objects and assume the availability of these implementations in the context of distributed applications. The existing implementations of proxy objects use their own specific way to generate the necessary code for the marshalling and demarschalling of messages that are forwarded between the proxies and the objects they act on behalf of.

The generation of GOAL interface definitions for proxy objects will not be possible without significant changes to the existing implementations of location independence. Rather then changing existing implementations of proxy objects the necessary infrastructure to support proxy objects can be added to the HLM migration mechanism as well.

The HLM migration mechanism will have to provide its own implementation of proxy objects as well as a supporting infrastructure for finding objects in the case of communication failures and for managing proxies within environments. This infrastructure has to be implemented for each participating environment. Interoperability with other implementations of location independence can be achieved on the level of remote method invocations.

Proxy Object Infrastructure

The implementation of an infrastructure for proxy objects in the context of the HLM migration mechanism has to be as simple as possible in order in be portable across all participating environments. In order to simplify the implementation the proxy infrastructure can be implemented as part of the `OM_Porter` interface.

The `OM_Porter` object of each participating environment can be used to locate sovereign objects, establish communication between proxy objects and sovereign objects, and will provide the necessary management for sovereign and proxy objects with in the particular environment. The `OM_Porter` object can be used to manage the location information of objects as well.

The `OM_Porter` will maintain a directory of all sovereign objects that exist within an environment, which can be called *sovereign directory*. Each sovereign object is required to register with the `OM_Porter` upon creation and will be assigned an object identifier that is also used to identify the sovereign object within the directory.

In order to maintain no more then one proxy per foreign sovereign object per environment the `OM_Porter` will also maintain a directory of proxy objects that exist within a particular environment which can be called *proxy directory*. For each proxy object the TCP/IP address and the object identifier of their sovereign object will be used to identify proxy objects within the directory.

The implementation proxy objects and sovereign objects can be generated for descendants of the `OM_Owner` and `OM_Migrateable` interfaces. In order to be able to forward messages to sovereign objects a proxy object needs to implement all signatures of their sovereign object and perform the necessary marshalling of messages including their parameters.

Based on an existing interface definition each signature can be redefined for the proxy object with the generated serialization code as a so called *proxy signature*. Each proxy signature will send a `send_remote_method_invocation()` message to the `OM_Porter` of the particular environment, which establishes and manages the network communication to the remote

`OM_Porter` of the sovereign object. Excerpt 4.b shows an interface definition and the corresponding proxy interface that can be generated.

```
interface My_Migrateable : OM_Migrateable {
OM_Boolean my_method(...){
     OM_Booolean ok;
     ...
     return(ok);
};
};

interface My_Migrateable_Proxy : My_Migrateable {
OM_Boolean my_method (...){
     OM_Boolean ok;
     // marshal parameters
     porter.send_remote_method_invocation(...);
     // demarshal result
     return(ok);
};
};
```

Excerpt 4.b: The proxy interface that can be generated automatically. In the above example the proxy signature `my_method()` is only shown schematically as the code to marshal the message and demarschal the result would be too long.

The `OM_Porter` establishes a network communication with the `OM_Porter` of the sovereign object based on the TCP/IP address that is stored by the proxy object. The object identifier of the sovereign object and the serialized message is send to the remote `OM_Porter` which performs a lookup of the object identifier in its sovereign directory and invokes the `handle_remote_method_invocation()` signature of the sovereign object.

Each sovereign object must be able to react to messages that are forwarded to it by its proxy objects. In order to do so an existing interface can be augmented automatically with a `handle_remote_method_invocation()` signature that contains the automatically generated code for the demarshalling of forwarded messages as well as the marshalling code for the return of the results. Excerpt 4.c shows the interface definition of excerpt 4.b and the signature that can be generated automatically to make it a sovereign signature

```
interface My_Migrateable : OM_Migrateable {
OM_Boolean my_method(...){
     OM_Booolean ok;
     ...
     return(ok);
};
OM_Boolean handle_remote_method_invocation(...){
     // demarshal message
     if (selektor = "my_method") {
          // demarshal parameters
          ok = this.my_method(...)
          // marshall return
          // return serialized result to porter
     };
     ...
};
};
```

Excerpt 4.c: The interface of excerpt 4.b can be augmented to become a sovereign interface through a signature that can be generated automatically. In the above example the signature `handle_remote_method_invocation()` is only shown schematically as the code to demarshal the message and marshal the result would be too long.

The `handle_remote_method_invocation()` signature will invoke the method that corresponds to the message of the original sender, i.e. the method `my_method()` in the example of excerpt 4.c. The return of this method invocation will be marshaled and the serialized result will be returned to the `OM_Porter` as the result of the invocation of the `handle_remote_method_invocation()` signature.

The `OM_Porter` will send the serialized result via the network connection back to the `OM_Porter` of the environment of the proxy object. That `OM_Porter` returns the still serialized result to the proxy objects. The proxy object demarshals the result and returns it to the sender of the original message.

Marshalling and Demarshalling

The marshalling and demarshalling code will generate and parse serialized representations of singular objects that are passed as parameters in messages to proxy objects or that are returned as results from methods invoked by sovereign objects. References that are passed as parameters or that are returned as results are managed in cooperation of the proxy and the sovereign object.

If a reference to a local sovereign or a local proxy object is passed in the message to a proxy object the marshalling code will determine the TCP/IP address and the object identifier of the corresponding sovereign object as well as the name of the corresponding proxy interface and will include all three in the serialized message.

The demarshalling code of the sovereign object the serialized message is forwarded to will determine if the TCP/IP address matches the one of its own environment. If so the `OM_Porter` is asked for a reference to the local object with the given object identifier, which the `OM_Porter` will retrieve from its sovereign directory.

If the TCP/IP address does not match the `OM_Porter` is asked for a reference to a proxy object for the given sovereign object. The `OM_Porter` will look into the proxy directory to determine whether a proxy already exists and will return a reference to the local proxy object if it does. If not the `OM_Porter` will create a proxy object from the proxy interface specified in the serialized message, add the new proxy object to the proxy directory and return a reference to it. The analogous operation is performed if necessary for the return of the result of the remote method invocation.

If the proxy interface specified within the serialized message is not available within the environment of the sovereign object the `OM_Porter` can ask the `OM_Porter` of the environment of the proxy object that send the remote method invocation to execute the negotiation algorithm of the HLM migration mechanism in order to determine the set of interfaces that need to be transferred in order to implement the missing proxy interface.

The need to transfer the interface definition for a proxy object may arise when an object of a newly created descendant interface is passed as a parameter in a message that is defined for the ancestor interface. Although legal in term of strong typing as the corresponding signature of the sovereign object can be compiled the implementation of the corresponding proxy may not be available within the environment of the sovereign object.

In addition to the basic operations described above the implementation of proxy objects in the context of the HLM migration mechanism has to be able to cope with several problems that may arise for the participating objects and environments. The problems include the inability to locate objects as well as the inability to contact the `OM_Porter` object of a participating environment.

Locating Sovereign Objects

During the execution of a remote method invocation the network connection that is used between the environment of the proxy object and the environment of the sovereign object may be lost due to communication failures. Such an event can be handled by the `OM_Porter` objects of both environments in a way that is transparent to both the proxy as well as the sovereign objects except for possible delays.

The `OM_Porter` that tries to communicate a serialized message or a serialized result respectively will retry to establish the communication once the network connection has been lost. The receiving `OM_Porter` can also verify a check-sum of the transferred data to determine whether the data has been corrupted while in transit. If so a retransmission can be requested.

More problematic are cases where the location information about a sovereign object stored by a proxy object, i.e. its TCP/IP address and object identifier is outdated. An `OM_Porter` that tries to establish a communication with a sovereign object via the corresponding `OM_Porter` object may receive an error-message that no object could be found for the given object identifier.

This is usually the case when the respective object has been migrated to another environment during the time since last forwarded message from the outdated proxy object. In order to prevent such a situation the `OM_Porter` object of the sovereign object can store the new TCP/IP address and object identifier of the migrated object that it receives as the update information during the release step of the completion phase of a migration.

The `OM_Porter` can maintain the entry with the new location information within the sovereign directory in order to provide a so called *forward pointer* to the sovereign object. The `OM_Porter` of the environment of the proxy object can then follow the forward pointer to find the sovereign object within another environment.

As an alternative a sovereign object can be required to know the locations of all its proxy objects in order to perform an immediate update as soon as the sovereign object is migrated. This operation will imply a high overhead with each migration operation while a some proxy may not be used during the time of several migrations of their sovereign objects. As the HLM migration mechanism is supposed to create as few dependencies between environments as possible, the former alternative is chosen.

Although feasible the use of forward pointer is not a complete solution to the problem, as the chain of forward pointers may break if just one directory entry is lost due to a malfunction of an environment. Forward pointers are also no help if one of the `OM_Porter` objects that maintain the chain of forward pointers is not available at the time a remote method invocation is attempted.

Directory Service

A more helpful solution would be the registration of sovereign objects with a federated directory service that maintains the location information on several hosts. This will not only lower the probability of total failure but will also enable a direct retrieval of update information about object that have migrated, provided the migration is registered with the directory service.

Host failures will no longer impede the operation of proxy objects as the an `OM_Porter` can access a different directory server if the default one fails and retrieve the necessary information from there. Furthermore, objects can be given symbolic names and a discovery of objects based on hierarchical namespaces can be performed as an alternative to direct location information with TCP/IP addresses and object identifiers.

Additionally the lifecycle of sovereign objects can also be managed with a federated name service. If a sovereign object is not only registered when it is created but also deregistered when it is deleted a “left-over” proxy object will be informed immediately that its sovereign object no longer exists.

Without the directory service and the lifecycle management a proxy object will learn that its sovereign objects has been deleted only after all forward pointers have been followed and the sovereign directory of the last `OM_Porter` of the forward chain does not contain an object for the given object identifier.

As an alternative proxy objects could be implemented to stay connected with their sovereign objects all the time. Despite the fact that a much higher consumptions of resources would result the resulting dependencies between environments would be prohibitive for some uses like migration to and from mobile devices. As the HLM migration mechanism is supposed to create as few dependencies between environments as possible, the former alternative is chosen.

Unfortunately the use of a federated name service will imply a significant higher effort for the implementation of proxy objects in the context of the HLM migration mechanism. The name service does not have to be a part of an implementation of the HLM migration mechanism but each participating environment will need to have access to at least one name server.

4.3 Extensions to the Migration Mechanism

Extensions to the HLM migration mechanism provide new features and capabilities for heterogeneous migration that question the objectives of the HLM migration mechanism. While extensions enlarge the applicability of the migration mechanism significantly they also place a high burden on the implementer of augmented versions of the HLM migration mechanism.

A great variety of extensions to the HLM migration mechanism can be envisioned but only the most promising examples in terms of their scope of applicability and ease of implementation are described here. Most of these extensions apply changes to object definitions and one requires changes to the participating environments.

The first subchapter describes an advanced migration techniques called adaptation that applies changes to object definitions during migration. This technique is used to implement several extensions of the HLM migration mechanism that are described in consecutive subchapters. The second subchapter outlines the use of adaptation to implement additional forms of negotiation.

The third subchapter describes fundamental changes to the migration algorithm of the HLM migration mechanism that are made possible through adaptation. The fourth subchapter finally explains how computations of objects can be transferred in the context of the HLM migration mechanism, through an extension that requires changes to the participating language environments.

4.3.1 Adaptive Migration

Several problems of heterogeneous migration that are not addressed by the HLM migration mechanism can not be solved without changes to the definitions of the objects to be migrated. If for example functionality that an object depends upon can not be added to the destination environment through an extension, migration will not be possible in the context of the HLM migration mechanism. If however the definition of the object to be migrated can be changed to use functionality that is available within the destination environment migration can be performed.

Changes to the definitions of objects will only be possible in simple well known cases as a general match between different object definitions can not be determined due to the problem of computability. In order to be able to work with differences of object definitions among environments new migration techniques need to be developed.

The HLM migration mechanism has to be able to determine certain properties of interface definitions which can be done by a technique that can be called *code analysis*. A comparison of interfaces based on these properties may reveal that an interface to be transferred to a destination environment needs to be changed.

The HLM migration mechanism must be able to apply changes to interface definitions through a technique that can be called *code adaptation*. Both of these techniques are described briefly in the following sections and several uses of these techniques are outlined in the subsequent subchapters

Code Analysis

The HLM migration mechanism uses only interface declarations to determine the dependent interfaces of the objects to be migrated. The method-code of the corresponding object definitions is not taken into consideration. An analysis of the method code can reveal new ways to perform a migration in situations where the dependent interfaces are not supported.

Objects almost never use the full functionality of other objects they reference as components. Therefore an analysis of the message passing expressions employed to access components can determine the subset of signatures of a dependent interface that is actually used. This

subset of the interface of an dependent object can be statically determined at compile time using techniques similar to liveliness analysis [AcS1997] and type inference [PaS1994].

While a general analysis of the semantics of an object is not possible due to the halting problem some simple questions about the actual usage of an object can nevertheless be answered. Whether for example a method of an object is used or not can be answered by searching for the corresponding message expressions within the application code.

The subset of the interface that is actually used can be called the *effective interface* with regard to its use as a component of an object. The effective interface can be used for example for new kinds of negotiations that determines whether the effective interfaces of dependent objects are supported by the destination environment. In order to be able to migrate an object under such circumstances definition of the dependent interface need to be changed accordingly.

Code Adaptation

An objective of the HLM migration mechanism states that the definition of objects involved in a migration are not changed by the migration mechanism. Changes to the corresponding interface definition of an object that is transferred during migration can nevertheless be used to overcome some limitations of the mechanism in special cases. Migration techniques that change interface definitions will be collectively called *code adaptation*.

Individual changes to interfaces definitions, also called *adaptations* reach from a simple exchange of interface names though the consistent renaming of signatures and message passing expressions to complex code changes like the generation of new signatures or whole new interfaces (see also the following subchapter).

Extensions to the HLM migration mechanism that use code adaptation will collectively be called *adaptive migration*. The advantage of adaptive migration is derived from the new ways to perform migrations in cases where the HLM migration mechanism falls short. For example the HLM migration mechanism will abort a migration when the interfaces necessary to migrate an object are neither supported nor extensible (see chapter 3 page 103). With adaptive migration other interfaces may be substituted through an appropriate change to the interface definitions that are transferred during migration.

Unfortunately, adaptive migration decreases the probability that the object semantics transferred during migration can be compiled within the destination environment. An adaptation may also cause the migrated objects to loose functionality or performance or both and the migration process may no longer be symmetric or transitive. The designer of the objects to be migrated has to decide whether such tradeoffs are worthwhile and what adaptations are acceptable under what circumstances. Being able to migrate objects at all will in most cases be the main reason to use adaptive migration.

While the idea of generating code during adaptive migration may appear unusual, examples of such techniques exist in other areas. Dynamic code generation at runtime is performed in the operating system Synthesis [Sch1996, PMI1988] for optimization purposes. Another example is the Exokernel [EKO1994] which extends the notion of dynamic code towards the underlying hardware-interfaces. Whole applications are generated for C++ in ERDoS [Cha1999] and for Java in Harness [MD+1998]. None of the approaches uses the technique for migration purposes though.

4.3.2 Variations of Negotiation

One of the characteristics of the HLM migration mechanism is that it compares interfaces only on the basis of their names which are supposed to be globally unique and unequivocal. With the ability to change the interface of an object to be migrated through adaptive migration other kinds of equivalence relations can be used.

In the general case the uniqueness of interface names can only be guaranteed when the designers of applications that use the HLM migration mechanism have access to a federated repository of interface definitions that is constantly maintained [WAA1998]. Since this is not likely to be the general case more sophisticated equivalence test will be necessary and naming conflicts that may arise need to be resolved.

A generic test for interface equivalence that is able to detect equal semantics of arbitrary named interfaces is not possible due to the halting problem. However, the structure of interfaces can be used for an partial test that is not just based on the name of the interface alone. All parts of an interface, its name, the name of the interface it inherits from, the names of interfaces of its components, the names of its signatures as well as the interface names of their parameters and their results can be part of such a test, that can be called *structural interface equivalence*.

In cases where the names of two interfaces or other elements of their structure do not match a measurement for their similarity can be provided on the basis of structural interface equivalence and can be used to find the best match for a specific interface within a whole set of different interfaces. Likewise, two equally named interfaces can be distinguished if their interface definitions do not match structurally.

Structural interface equivalence is by no means an ideal tool because interfaces with identical structures may still implement different semantics[25]. The ability to detect that interfaces do not match may nevertheless be high enough for practical purposes especially if names of standard interfaces are involved. The structural interface equivalence can be used to both detect naming conflicts as well as similarities among interfaces as exemplified in the following sections.

Interface Renaming

Using structural interface equivalence the negotiation algorithm of the HLM migration mechanism will be able to detect if an interface supported by a destination environment bears the same name but has a different structure than an interface that is used by an object to be migrated. The resulting *name conflict* can be resolved through a consistent renaming of all occurrences of the ambiguous interface name in question within all interfaces of the interface set to an arbitrary interface name prior to the negotiation process.

This technique can be called *interface renaming* and it enables the transfer of the interface in question as well as its use within the destination environment provided that the renamed interface is migrateable, i.e. all interfaces it depends upon are either supported or extensible. The renaming of interfaces represents the simplest form of adaptive migration. Consistent class renaming is for example performed in OZ [NN+1998] in the context of homogeneous migration.

In order to support consecutive or reverse migrations the renamings applied need to be stored by both environments in order to simplify further migrations. Structural interface equivalence will be able to detect a renaming due to a naming conflict, i.e. two interfaces match in their structure but not in their name. Whether a reverse renaming of interface names can be performed automatically in such a situation depends on the individual case.

Interface Mapping

If a dependent interface is not supported and it is also not extensible in the context of the destination environment the HLM migration mechanism will abort. Using structural equivalence and adaptive migration another interface may be used to replace the missing one in order to continue the migration. All migration techniques that map one interface to another will collectively be called *interface mapping*[26].

The following two interface mapping techniques can be used as extensions to the HLM migration mechanism when the normal negotiation algorithm is not successful. These techniques aim to determine a surrogate interface if a dependent interface is not available. Additional mappings can be defined as the two examples of this techniques described here do not provide a complete list of possible variations of interface mapping.

Interface Reduction

Probably the simplest example of an interface mapping technique is the comparison of an interface against its ancestor. Based on structural interface equivalence and the effective interface of an object a match of an interface against its ancestor may reveal that only

[25] Also, two interfaces with the same name but different structure may still implement the same semantics.
[26] Interface mapping uses interface renaming to replace an interface name with a specific other one.

signatures inherited from the ancestor interface are actually used. The ancestor interface can then be used in place of the interface in question.

This test may be extended to a comparison of the transitive closure of the inheritance relation of the interface in question. What interfaces further up in the inheritance tree are still acceptable depends upon whether signatures that have been redefined or added are used by the effective interface. As a general rule only the most specific interfaces of the inheritance chain with regard to the effective interface can be used.

If a matching interface is found it can be inserted into all relevant interface definitions of a negotiation set in place of the unsupported one using adaptive migration. Because this technique effectively reduces the need for a specific interface to the use of its ancestor interface this interface mapping technique can be called *interface reduction*.

An example of interface reduction may be a migration of an object that uses a double linked list which extends the interface of a single linked list. A code analysis of the use of the list may reveal that the effective interface includes only signatures defined by the interface of the single linked list which may therefore be used instead.

Interface Substitution

In the general case an unsupported interface may be substituted with a different one if the effective interface of the unsupported one is structural equivalent to the different one. This condition ensures that at least every signature used can be matched with a method that can be invoked. The new interface can be substituted in place of the old using consistent renaming. This interface mapping technique can be called *interface substitution*.

The probability that an object will still work as intended when it is recreated within the destination environment with a structural equivalent but differently implemented interface depends upon the individual case. As this question is undecidable in general due to the halting problem, heuristics have to be developed that can be used by the designers of applications of migration to decide at runtime whether a migration using interface substitution should be attempted or not.

In addition to the measurement of the similarity of interfaces that can be developed, preconditions for the substitution of interfaces can be defined by the developer of an object for the dependent interfaces used. E.g. certain interfaces can not be substituted, other can only be reduced etc. The possible combinations of interface mapping techniques to define new forms of negotiation are virtually limitless; yet these techniques have to be tried intensively in practical scenarios to assess their applicability.

Similar techniques to the ones described above have been proposed for homogeneous migration. For example OZ++ [TH+1995] combines different versions of classes of a multiple inheritance lattice during migration. A dynamic load technique has been developed by Acharya and Saltz [AcS1996] that allows the integration of functionality of the destination environment into a migrated object. Last not least, type conformity among abstract types is defined in Emerald [RT+1991] but not used during migration, as a homogeneous environment is assumed.

4.3.3 Informed Migration

The determination of the migration set within the HLM migration mechanism is performed without knowledge about the destination environment. The root object of a migration request as well as its related objects can determine which objects they depend upon only on the basis of their state and their relationships. This form of migration can be characterized as *uninformed migration*.

If the set of interfaces supported by the destination environment is available during the determination of the migration set the objects involved can use that information for their decision about which objects they depend upon. The use of knowledge about the destination environment within the migration process can be characterized as *informed migration*.

Despite the fact that knowledge about the destination environment can be gathered early in the migration process, informed migration can only be implemented in the context of adaptive

migration. An informed decision of an `OM_Migrateable` object not to depend on a component whose interface is not supported makes only sense if the interface definition of that object can be changed accordingly.

The HLM migration mechanism can be extended with a second negotiation phase that uses informed migration and adaptation. The normal negotiation algorithm of the HLM migration mechanism may not be able to identify supported interfaces or extensible interfaces for all dependent interfaces of a given migration set.

A second negotiation phase using informed decisions of the `OM_Migrateable` objects about their dependent objects can be used to reduce the migration set in such a way that the set of dependent interfaces is included in the union of the set of supported interfaces and the set of extensible ones. The interfaces of the objects of the migration set will then have to be changed accordingly using adaptive migration.

For example a designer of a migrateable object may use an interface that implements a balanced tree for its implementation. Using informed migration he can also implement a fallback to the interface of a sorted list that is used if the interface of the optimized tree is not available within a destination environment.

The combination of informed and adaptive migration can even be used to merge the determination of the migration set and the determination of the interface set into a single phase. Using the knowledge about the destination environment the objects to be migrated can find out themselves which of their dependent interfaces are supported or extensible and can indicate the appropriate adaptations to their object definitions. Such a significant change to the migration algorithm of the HLM migration mechanism can lead to new forms of migration.

Fragmented Migration

With the above combination of informed migration and adaptive migration a migrateable object can be designed to split itself if only a part of its behavior is supported within the destination environment. If an object can not be migrated because some of its dependent interfaces are not supported the designer can enable the object to manipulate its interface definition and representation in such a way that the unsupported parts of the object are left at the source and only a so called *fragment* of the object whose interfaces are supported is migrated.

A new interface definition for the supported part will have to be generated using adaptive migration and only the corresponding part of the state of the object needs to be represented. The remaining part of the object will have to be designed with conditional behavior as only a subset of its state will be accessible locally after migration. Alternatively both the remaining and the migrated fragment of an object that is split in this way may use appropriately generated proxy objects to communicate and exchange state changes in order to maintain a shared state. The transfer of only a fragment of an object can be called *fragmented migration*.

The migration of only a fragment of the original object touches a design related question. It marks the opportunity to split an object into a migrateable and non-migrateable part. Being able to decide this question not statically at design time but at the time of migration based on the availability of interfaces and the actual state of the object and being able to combine this decision with the possibility to share state among the fragments leads to a novel mix of object migration and object replication.

In the context of fragmented migration the behavior of an object will have to be designed with defined joints. The fragments will either share the identity of the object they have been created from and will have to be managed like replicas or a new identity will have to be created for either the remaining or the migrated fragment. As a consequence the fragments will communicate like replicas using a synchronization protocol or work completely independent once the supported fragment has been migrated.

During the course of several migrations the need of the reverse operation to fragmentation may also appear. When fragments of an object that have been created through previous fragmented migration happen to be migrated to the same system again a reverse merge of the fragments to a single object may be performed. If an object can be split along various joints and several

consecutive fragmented migrations and mergers are performed arbitrary combinations of object fragments may result at least theoretically.

The use of fragments has been implemented for homogeneous migration in SOS [SG+1989] where fragments of objects are defined statically as special kinds of proxies. Fragments are also implemented in HADAS [HoB1997] and HERON [FJ+1992] as well as in combination with replication in AscpetIX [GS+1998]. The agent system Open Services Model [LG+1997] also defines split and sync operations for the implementation of mobile agents.

4.3.4 Heterogeneous Migration of Computations

The HLM migration mechanism does not support the transfer of computations of objects among existing heterogeneous language environments. This problem has been excluded from the design of the mechanism because it can not be addressed without changes to the participating environments prior to migration.

The migration of computations of an object involves the transfer of the activation records of the object to be migrated in addition to its identity, functionality, and state. An activation record is created for each invocation of a method of an object and is stored on a so called *stack* of activation records or *stack frames*.

A stack of activation records together with a stack pointer and program counter are called a *thread*. A thread represents the execution of the semantics by the processing hardware. The flow of control among objects manifests itself as a stack of activation records and the context of the execution at any point in time is defined by the stack pointer and the program counter that refer to the current invocation as well as to the current instruction to be executed respectively.

The migration of computations of objects requires the transfer of all sequences of activation records that have been invoked for the objects to be migrated to the destination environment. The transfer of these so called *stack segments* is necessary in order to be able to continue the execution after migration has been performed.

When the flow of control returns to an activation record the code of the methods that have been invoked and the state of the objects the method have been invoked for have to be available within the same execution environment. As the objects, i.e. their state and their behavior in the form of methods are migrated the corresponding activation records have to be transferred to.

As a consequence, the thread or more precisely the stack of activation records becomes distributed and the flow of control needs to be passed back and forth between the environments in order to continue the computation. New invocations of methods may occur within both environments, but the flow of control will have to move between environments depending where the object whose methods are invoked reside. Figure 4.j: illustrates the transfer of activation records and the distributed flow of control that results.

The HLM migration mechanism initiates migration synchronously, i.e. via method invocation and is supposed to work for single threaded environments. As a consequence the execution of the migration algorithm will happen at the top of the stack while the activation records of the objects to be migrated have to be extracted from lower portions of the same stack (see also chapter 3 page 99).

In contrast multi-threaded environments enable the execution of several more or less independent threads in parallel and complete threads including all activation records can be migrated among homogeneous environments through an operation called *thread migration*. As a consequence of thread migration the objects the activation records depend upon have to be transferred as well during thread migration or have to be migrated to the place of execution as necessary during the continued execution of the thread (see also page 203).

In the context of the HLM migration mechanism only the relevant segments of a single thread are transferred and in all cases additional activation records of previous or subsequent invocations will remain at the source. As a consequence proxy objects have to be used for the references of activation records that refer to objects of the other environment.

Additionally the stacks of activation records at both source and destination environments have to be modified in such a way as if remote method invocations between the object of the source and the objects migrated to the destination had occurred in the first place. These modifications of the stack of activations records can not be performed without the cooperation of the participating environments.

The transfer of computations of objects can be regarded as the reverse operation to remote method invocation as not an new activation record is created within a remote environment in order to move the flow of control to a remote object but an existing object and its activation records are moved to a remote environment and flow of control will follow when it comes around to return to theses activation records.

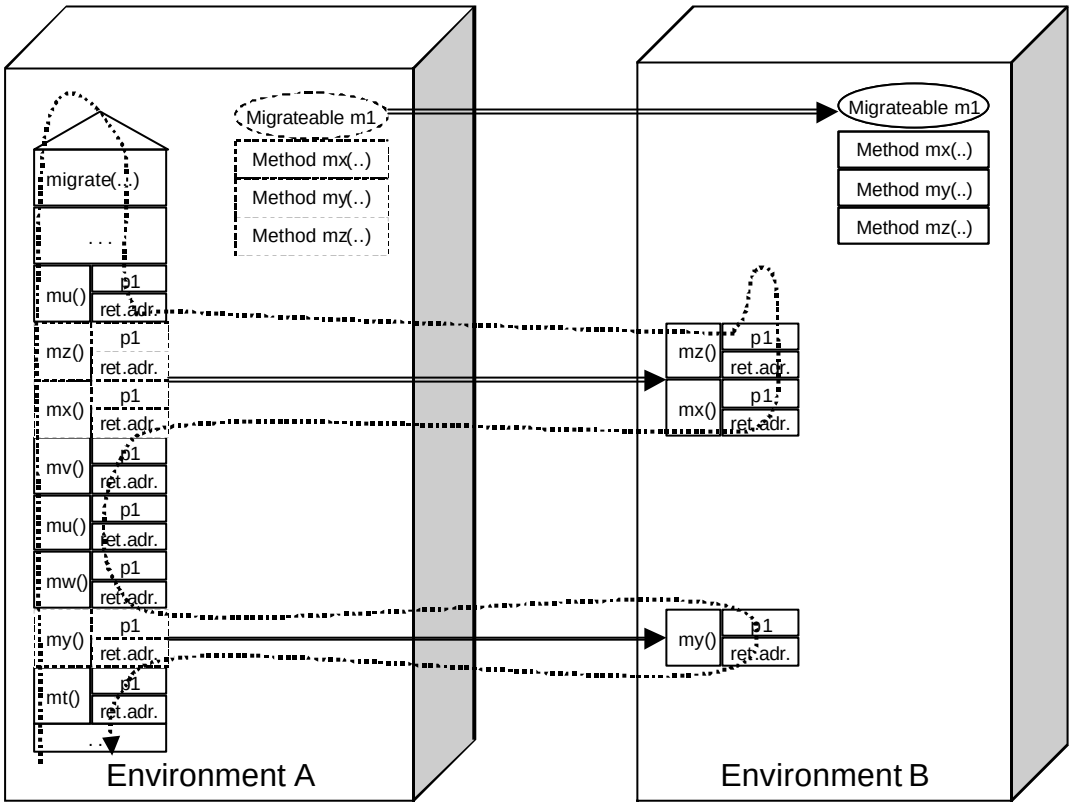

Figure 4.j: The migration of computations of an object requires the transfer of the activation records that have been created for invocations of the methods of the object. The flow of control will have to move between the source and the destination environment in order to continue the computations. In the above example the activation records for the methods `mx()`, `my()` and `mz()` of object m1 are transferred with the state and the methods of object m1 to environment B. The flow of control (shown here as a dotted line) will have to return after the migration has been performed to the transferred activation records and additional invocations may occur throughout the course of the continued computation (the various references are not shown here in order to avoid confusion).

In order to implement the transfer of computations in the context of the HLM migration mechanism an additional “transfer of computations” phase has to be performed by the migration algorithm after the transfer of state phase. Additionally the completion phase of the migration algorithm has to be modified.

At that stage in the migration process the semantics of the object to be migrated as well as the state of objects including the proxy objects of their related objects of the source environment will have been established within the destination environment. Excerpt 4.d shows a code fragment that is used throughout the rest of the chapter to illustrate the necessary operation of the transfer of computations phase.

```
interface K : OM_Owner {
...
OM_Boolean mk(M pm,...) {
OM_Boolean ok;
ok = pm.mm(n1,...);                                          //1
return(ok);};
};
interface M : OM_Migrateable {
...
OM_Boolean mm(N pn,...){
OM_Boolean ok;
ok = pn.mn(o1,...);                                          //2
return(ok);};
};
interface N : OM_Owner {
...
OM_Boolean mn(O po,...) {
OM_Boolean ok;
ok = po.mo(p1);                                              //3
return (ok); };
};
interface O : OM_Object {
...
OM_Boolean mo(P pp, ...){
OM_Boolean ok;
ok = pp.mp(...);                                             //4
return (ok); };
};
interface P : OM_Object {
OM_Boolean mp(...){
...
return (true); };
};
```

Excerpt 4.b: Five invocations of methods of five different interfaces are used as the example to illustrate the transfer of computations in the context of the HLM migration mechanism. The chain of invocations starts with the method `mk()` of the interface `K` that takes a parameter `pm` of interface `M` and send it the message `mm()` with the actual parameter `p1` (see `//1`). The other invocations follow the same scheme (see `//2` through `//4` methods are named with a prefix `m` while parameters are named a the prefix `p`, the declarations of the related objects that are passed as actual parameters as well as additional lines of code are not shown for reasons of space).

The transfer of computations phase performs the following steps: the relevant stack segments are identified and transferred, the references of all activation records are substituted, stack frames for the remote method invocations are inserted and finally the return addresses are redirected. These steps can not be performed without the availability of the necessary functionality within the participating environments.

A. Identification and Transfer

The first step of the transfer of computations phase is the identification of the relevant stack segments as well as their transfer. One or several consecutive invocations of methods of objects of the migration set form a relevant stack segment. Each stack segment that is transferred also has to include the activation record of the next invocation as this will be needed later for the insertion of the corresponding remote method invocation.

The identification of the relevant stack segments can not be performed without the availability of the necessary information within the source environment. Implementations of object-based

languages usually include a reference to the receiver of a message within the activation record or the corresponding method invocation. However no information about the layout of activation records is included on the stack or in the executable code.

In order to enable the identification of the relevant activation records a language environment does not only have to provide means of access to activation records but also the information about their layout. Since a traversal of the whole stack would still be necessary a language environment can also chain activation records of each object during execution.

The relevant activation records can then be collected easily through a traversal of the references that point from one activation record of an object to the next one which may appear many invocations later on the stack. Unfortunately such a chaining of activations records can have significant performance implication [JL+1988]. Figure 4.k shows the situation of the example of excerpt 4.b prior to migration.

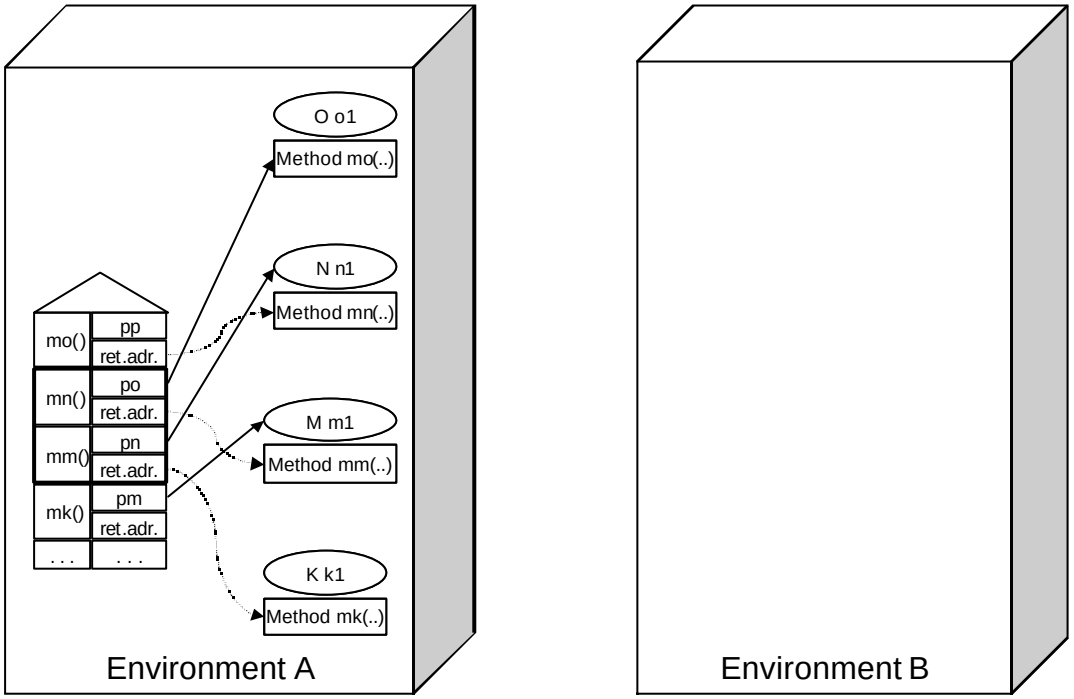

Figure 4.k: The situation of excerpt 4.b before object `m1` will be migrated. A fragment of the execution stack is shown as well as four objects that are used as parameters and receivers of messages respectively. The stack segment (shown here as a bold rectangle) that needs to be transferred for object `m1` consists of the invocation of method `mm()` as well as of the subsequent invocation of method `mn()`. (References of activation records to objects are shown as arrows, curved dotted arrows are used to link return addresses with the methods they point to, only the most relevant references are shown to avoid confusion and additional parameters have been omitted)

The relevant stack segments have to be transferred to the destination environment in an appropriate format that contains the following information: the contents of activation record that include the actual parameters, a reference to the object the method was invoked for as well as the return address, the name of the method invoked, and the name of the interface of the object the method was invoked for.

The actual parameters of the activation record including the reference to the object the method was invoked for have to be represented in a similar way as within serialized messages of remote method invocations in order to allow the substitution of references (see page 158). The return address contained within the activation records has to be transferred in a special format in order to allow its redirection. The name of the method that have been invoked and the name of the interface of the object the method was invoked for can be represented textually.

While all activation records of each relevant stack segment are transferred to the destination, the first and the last activation record, in the order of invocation, of each stack segment are maintained within the source environment. These activation records will be reused for the remote method invocations that will be added in the insertion step.

B. Substitution

As a second step, both the source and the destination need to substitute the references of activation records that either remain at the source but refer to migrated objects or that are transferred to the destination but refer to objects that remain at the source. The invalid references have to be replaced with references to corresponding proxy objects that have to be created if they not already exist within the respective environments.

Within the source environment references of remaining activation records to migrated objects have to be substituted with references to the corresponding proxy objects. At this point in time the state of these objects has just been migrated and the proxy objects will have to be created anew, effectively obviating their creation during the completion phase.

Within the destination environment references from activation records that have been transferred to objects that remains at the source have to be substituted with references to corresponding proxy objects. If the objects referred to are also related to the migrated objects their proxies haven been created during the transfer of state phase; if not they have to be created now. Figure 4.l shows the example of figure 4.k after substitution of references has been performed.

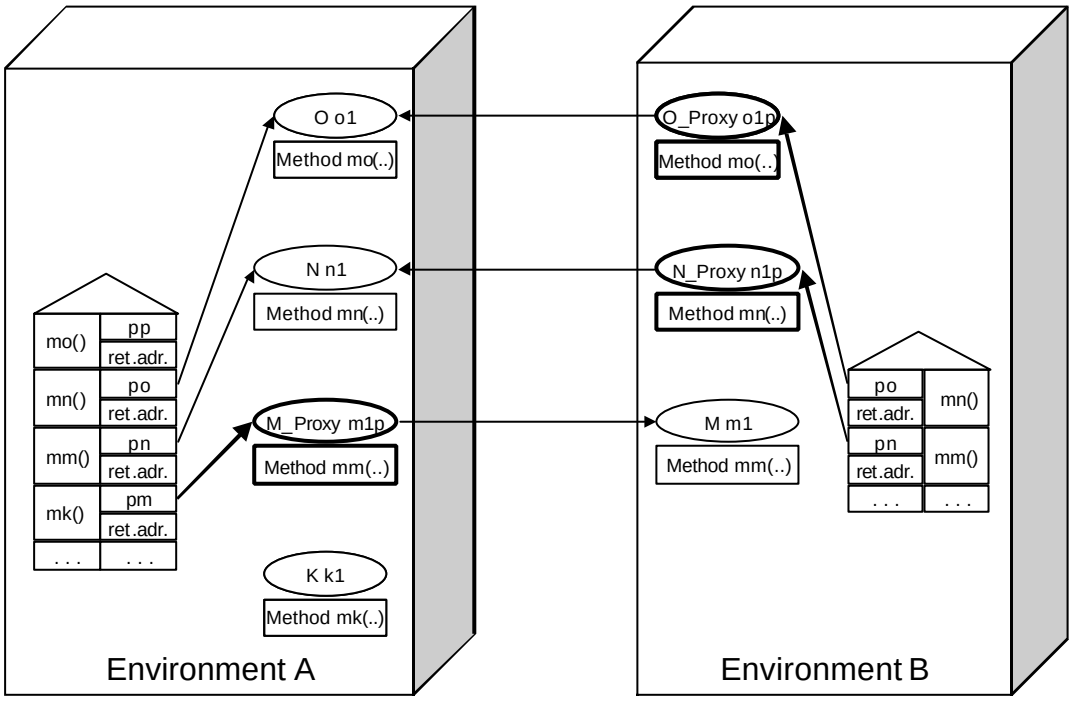

Figure 4.k: The situation of the example of excerpt 4.b after the substitution of references has been performed. Object `m1` has been migrated to environment `B` including the relevant stack segments. The proxy object `m1p` has been created in environment `A` and the reference of the parameter `pm` of the invocation of method `mk()` has been substituted. Within environment `B` the proxy objects `o1p` and `n1p` have been created and the references of the transferred activation records have been substituted accordingly (newly created object as well as substituted references are shown here as bold lines and arrows respectively, only the most relevant references are shown to avoid confusion and additional parameters as well as links of return addresses have been omitted).

Migrated activation records may also contain singular objects that are passed as parameters. These singular objects have to be converted accordingly if a binary format is used and the internal representations of source and destination environments differ. The ORL representation can be used as a common ground for the transfer as well.

In the context of the HLM migration mechanism access to local state of object is only possible through the invocation of methods. Therefore no references to internals of objects can exist within activation records. Unfortunately optimization techniques applied by modern language compilers may invalidate that assertion and have to be handled specifically.

The substitution of references of existing activation records within the source environment and the creation of new activation records within the destination environment can not be performed without the availability of the necessary functionality to create and manipulate activation records within the participating language environments.

C. Insertion

As a third step, the now distributed stack segments will be arranged as if remote method invocations among the environments had been performed in the first place. In order to do so existing activation records will be remapped to methods of proxy and sovereign objects as appropriate and additional activation records will be inserted for the management of remote method invocations as necessary.

In the original sequence of events local method invocations of objects of the source environment have been performed and have led to the migration that is being performed. As part of the migration activation records that are awaiting the return of the flow of control are transferred as part of the migration of their objects.

The migration process has to create the “missing” activation records that are necessary to let the now distributed stack segments appear to be the result of remote method invocations. Existing activation records have to be changed and new stack frames have to be created to achieve this metamorphosis.

The original invocations of methods of migrated objects are mapped to invocations of the equivalent methods of the corresponding proxy objects. After each invocation of methods of proxy objects of the migrated objects an additional activation record is inserted for the necessary invocation of the `send_remote_method_invocation()` method of the `OM_Porter` object.

At the destination environment additional invocations of the `handle_rmi()` method of the `OM_Porter` and of the `send_remote_method_invocation()` method of the migrated objects are inserted for each transferred stack segment. The original invocations of methods of object of the source environment by migrated objects are transferred as part of the stack segments and are mapped to invocations of the equivalent method of proxy objects of the destination. After each invocation of methods of proxy objects of objects of the source an additional activation record is inserted at the destination for the necessary invocation of the `send_remote_method_invocation()` method of the `OM_Porter` object.

The original invocations of methods of object of the source by the migrated objects are maintained but activation records for the invocations of the `handle_rmi()` method of the `OM_Porter` and of the `send_remote_method_invocation()` method of the object of the source are inserted in each case.

The newly created activation records are created with the appropriate parameters that are known to the respective `OM_Porter` objects. The references to the receivers as well as the TCP/IP addresses and object identifier of sovereign and proxy objects can be determined by the `OM_Porter` objects easily. Figure 4.l shows the example of excerpt 4.b after the insertion of activation records has been performed.

The insertion of additional activation records is not possible without availability of necessary functionality within the participating language environments. New activation records have to be

created within the existing stack of activation records of the source environment or outside of the stack within the destination environment, to be inserted later.

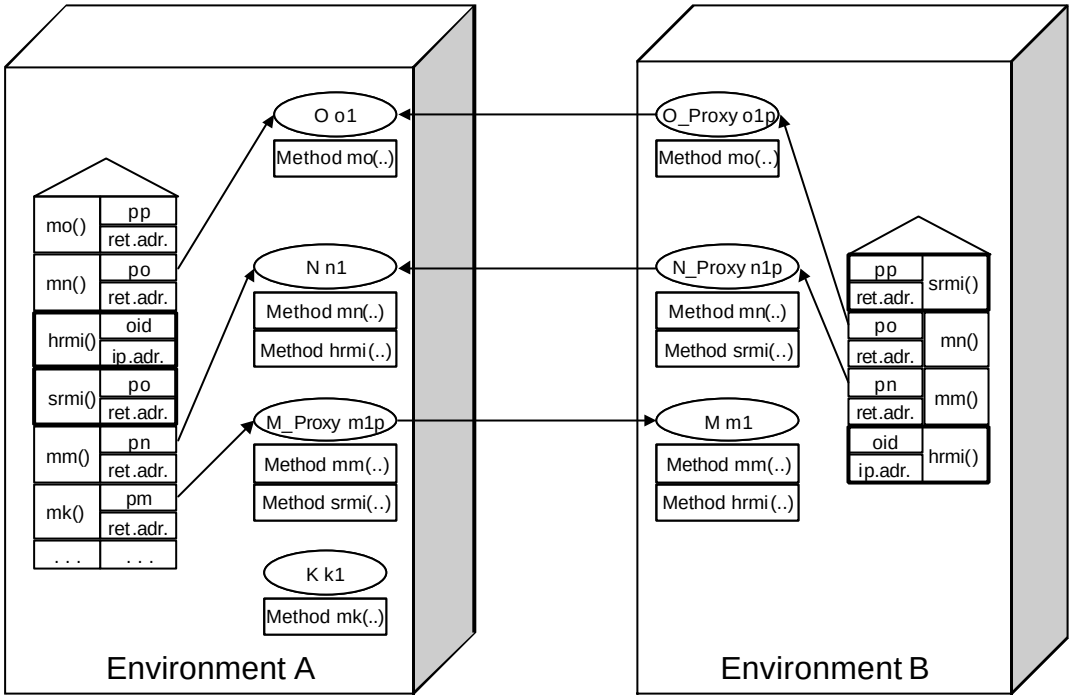

Figure 4.l: The situation of the example of excerpt 4.b after the substitution of references has been performed. For each proxy object a pair of invocations of the `send_rmi()` and a `handle_rmi()` methods of the corresponding `OM_Porter` has to be inserted into the distributed stack fragments (newly created activation records are shown here with bold lines, for reasons of space the method names have been abbreviated and not all activation records that have to be created are shown, only the most relevant references are shown to avoid confusion and additional parameters as well as links of return addresses have been omitted).

The implementation of proxy objects in the context of the HLM migration mechanism encapsulates the marshalling and demarshalling of messages and results within proxy and sovereign objects and encapsulates the network connection management within the `OM_Porter` interface. This separation enables the recreation of remote method invocations that have not happened in the first place.

D. Redirection

As a fourth step the return addresses of the activation records have to be redirected as the corresponding methods have been migrated or the activation records have to be remapped to proxy objects. The determination of the correct return addresses is not possible without the preparation of the necessary information by the participating language environments.

As proxy objects have been substituted for the migrated objects as well as for the objects of the source that need to be referenced from the destination environment the return addresses of the respective activation records will have to be redirected to the instruction after the forwarding of the corresponding message returns. All possible return addresses for this case can be determined statically during the generation of the method code for the proxy interfaces.

Invocations of methods of objects that are used as sovereign objects after migration by their newly created proxies have to be redirected to the corresponding invocations by their `handle_remote_method_invocation()` methods instead of returning locally. All possible return addresses for this case can be determined statically during the generation of the method code for the sovereign interfaces.

Additionally, the return addresses of activation records that are transferred to the destination for migrated objects are no longer valid. For each of these activation records the return address within the source environment has to be matched to a symbolic value and the corresponding method invocation has to be identified for the newly generated method code within the destination environment. Figure 4.m shows the example of excerpt 4.b after the redirection of return addresses.

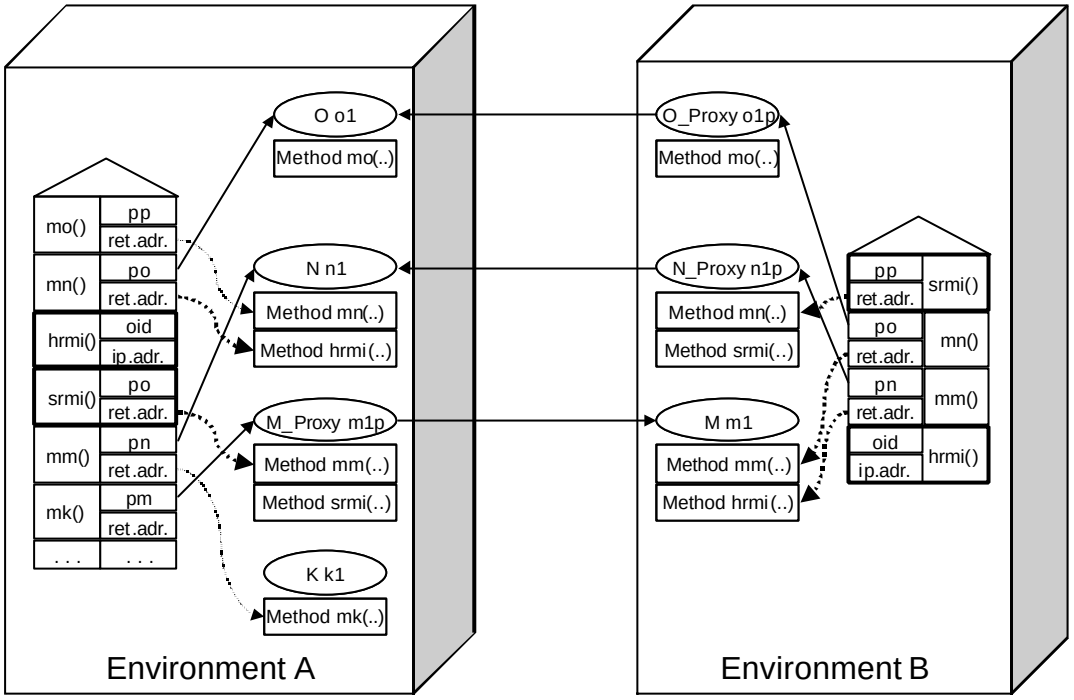

Figure 4.m: The example of excerpt 4.b after the redirection of return addresses has been performed. The return of the flow of control "to" proxy objects as well as "to" sovereign objects has to be determined anew including the handling of network communications through the `OM_Porter` (not shown here; redirected return addresses are shown here as bold dotted arrows, only the most relevant references are shown to avoid confusion and additional parameters have been omitted).

Redirection of return addresses are only possible if both source and destination environments provide enough symbolic information as an augmentation of the executable method code that a match of the execution state on a message by message basis can be performed by the `OM_Porter` that processes the migration.

Completion

If the transfer of computations phase has been successful the migration algorithm of the HLM migration mechanism can perform the completion phase in the same way as before. If an commit is performed some proxy objects are already available within the source environment as they have been created by the substitution step of the transfer of computations phase.

After the migration has been completed successfully, the newly created stack segment within the destination environment have to be moved to the top of the stack and the flow of control is redirected to that part of the `send_rmi()` method of the `OM_Porter` that waits for a the communication of the result.

This manipulation of the stack and redirection of the flow of control is necessary to ensure the migrated computations are executed correctly within single-threaded environments. The transferred stack segment will be executed in place of the `handle_migration()` activation record that performed the destination part of the migration. The flow of control of the destination will return to the invocations prior to the migration after the migrated computations of the object have been completed.

The manipulation of the stack and of the program counter can not be performed without the availability of the necessary functionality within the destination environment. These operation are especially delicate as they have to be performed without the creation of new activation records on the stack.

For the execution of the artificial remote method invocations between the source and destination environments the `OM_Porter` objects have to be able to reestablish communications for the return of the serialized results as if network connection hand been disrupted in the meantime.

The support of the transfer of computations in the context of the HLM migration mechanism requires the enabling of the participating languages environments with the necessary services prior to migration. The enabling approach to overcome heterogeneity can be used in addition to the restricting, conversion and extension approaches used by the HLM migration mechanism (see also chapter 2 page 60).

The functionality the environments have to be enabled with comprises the identification, access and manipulation of activation records, the shift of whole portions of a stack, the creation of new activation records, the replacement of the topmost stack frame as well as the redirection of the program counter.

Furthermore, the compiler of the participating environments has to provide enough information to perform all these operations as well as to determine the location of return addresses within the generated method code based on message expressions that are specified by the corresponding interface definitions.

The implementation of this enabling functionality requires significant if not fundamental changes to the participating language environments. Fortunately the enabling functionality can be implemented independently of the particular programming languages, i.e. the language definitions do not have to changed.

Problems with Heterogeneous Computations

In addition to the significant implementation effort that is required in order to provide the necessary enabling functionality, the support for the transfer of computations in the context of the HLM migration mechanism will also lead to a number of unforeseen problems that have to be addressed.

Code Optimization

The necessary identification and extraction of activation records can become prohibitively complex if sophisticated code optimization techniques are used. The use of register allocation, or code movement can have detrimental effects on the implementation transfer of computations as not all information about invocations may be saved in the corresponding activation records.

Information that is kept in registers of the central processing unit may be saved in different activation records if for example the register allocation for that invocation runs out of registers. The information that should be kept within a single activation record may be scattered among various records or may not be stored in activation records at all.

The movement of code across method invocations that is not unusual with wide-spread superscalar processor architectures may undermine the assertion that a message passing expression within the source code can be mapped to an invocation of a new stack frame within the native machine code.

Although some techniques have been proposed to address the increasing semantic gap between high level languages and highly optimized machine code no practical solution for this problem is available. The use of sophisticated code-optimization techniques will likely prohibit the implementation of the transfer of computations in some cases.

Multi-threading

Unfortunately, the proxy-based implementation of support for the transfer of computations does not make much sense without support for multi-threading. The prototypical implementation of

the HLM migration mechanism implements a rendezvous of the `OM_Porter` objects of two environments that perform the migration and continued with their single-threaded operations afterwards. TCP/IP sockets are used to perform the rendezvous.

The support of proxy objects in the context of the HLM migration mechanism introduces a dependence between the environments as remote method invocations have to be alternated with migration requests. The confinement to a single thread deteriorates the performance of the remote method invocations as only single request can be handled at a time.

The support of transfer of computations in the context of the HLM migration mechanism requires that computations of the destination are suspended until the migrated computations have been finished. This creates an almost unbearable situation that becomes a deadlock, if a remote method invocation to the destination has occurred between the last migrated computation and the migration request.

A much better solution in the context of multi-threaded environments uses a separate thread to wait for incoming network connections and creates a new tread to handle each remote method invocation. Unfortunately only few existing language environments support multi-threading natively and even worse as the HLM migration mechanism is supposed to be working symmetrically all environments will have to support multithreading.

Heterogeneous migration of computations within multi-threaded environments becomes more complicate as an object may take part in several independent computations. The activation records of several threads will have to be transferred accordingly. The control of the access to the state of the object through the usual synchronization primitives, like semaphores or monitors, will have to be performed in a distributed fashion and the prevention of deadlocks will become very complex.

Other Techniques

Several different techniques exist for the migration of computations of objects or whole threads among homogeneous environments. These techniques have not been chosen as extensions of the HLM migration mechanism due to their limited support for heterogeneity as well as various specific problems.

Distributed computations have been implemented successfully by Emerald [JH+1988] for homogeneous environments using transparent remote references. OS Emerald [StJ1995] extends this support to heterogeneous hardware using the conversion approach. The implementation benefits from the fact that activation records are managed within Emerald in the same way as migrateable objects. The benefits of this technique are described by Jul et al. [JL+1988] as follows :

> “Moving invocation frames along with the objects in which they execute ensures that execution can continue as long as possible and removes the computational burden from nodes that do not need to be involved in communications”

A similar technique is used for homogeneous thread migration in the agent system Sumatra [ARS1996] that extend the Java virtual machine to be able to save the computational state. The transfer of computational state is implemented at the source code level in the programming language Beta [BrM1993] that features activation records as first class objects, i.e. patterns.

A number of source based approaches to the migration of computational state have been proposed on the basis of source code transformations. In the WASP project [Fue1998] the Java exception mechanism is used to initiate migration and to unwind the stack of activation records through state saving operations within an exception handler for each method. The computational state is recreated at the destination using generated conditions within the method code.

Code transformations are also used in Arachne [DiR1998] to enable heterogeneous thread migration in the context of a number of additional source code restrictions. The programming language Borg [BeH2000] enables the reification of computational state through the use of explicit push statements.

All source code based approaches share a number of disadvantages. The necessary transformation of source code through preprocessors prohibits the use of source code debugging tools as the code written by the developer is changed significantly prior to compilation. The additional operations necessary may also slow down the execution of the generated native code to some extend.

As a less problematic alternative a similar instrumentation of Java byte code has also been proposed [CT+2000]. This technique is able to capture activation records directly through the augmentation of the compiled code of methods. Although independent of source code changes this approach also impedes the use of debugging tools.

4.4 Additional Language Concepts

The HLM migration mechanism is able to migrate objects between existing language environments through the restriction of the design of the objects to a common set of concepts, namely encapsulation of state and behavior, single inheritance and synchronous message passing. Environments that offer additional concepts can still participate in migration if the objects to be migrated are designed and implemented using only these supported concepts.

Support for additional language concepts can be added to the HLM migration mechanism through various extensions but the necessary design changes will differ depending on the particular concept to be added. The implementation effort will also vary with the participating environments. As concept may be added easily to some environments but very hard to others.

A concept is often difficult and in some cases impossible to address by migration if an object that uses the concept has to be migrated to an environment that does not natively support the concept. Because some concepts are even contradictory equivalent support by all environments as well as support of any combination of concepts will not be possible in all cases.

The effort that is necessary to support a particular concept may reach from the implementation of an additional standard interfaces or minor enhancements of the HLM migration mechanism to adaptive changes of the objects being migrated during migration as well as to enabling changes of the participating environments prior to migration.

The *grade of migrateability* of objects can be determined in terms of the effort necessary to support a particular combination of concepts used by an objects in the context of particular source and destination environments. As the support for a concept for a particular combination of environments may require relative high efforts in one and comparatively low efforts in the other direction migration can also be characterized as one way in this regard.

The following subchapters discuss what techniques can be used to support some prominent language concepts in the context of HLM migration mechanism. Each subchapter discusses a group of related concepts but the list of concepts is not complete as not all existing concepts and environments can be covered. Only the concepts with the greatest practical relevance have been chosen. An overview of principle migrateability between existing language environments concludes this chapter.

As the conditions of migration my vary significantly depending on the concepts supported by the participating environments, each concept is discussed separately. The effort to support a particular concept may also differ significantly whether the concept is available at the source or at the destination or at both sides but in different form.

In order to distinguish each particular case a separate headline that denotes the characteristics of the source and destination environment is used in the following subchapters wherever appropriate. Within these headline, the direction of migration is indicated through the source environment on the left and the destination environment on the right and an greater-or-equal => sign indicates that migration can be performed on the basis of the HLM migration mechanism.

Parentheses around the greater-or-equal (=>) sign indicate that migration can not be performed in all cases using the HLM migration mechanism as it has to be enhanced. Parentheses around a greater (>) sign indicate that significant extensions to the HLM migration mechanism will be required or to the participating environments.

4.4.1 Type Systems

The majority of programming languages uses the notion of *types* to classify data and operations. A type defines the format that is used to represent data of that type and it defines the operations that can be applied to the data. A language is said to be *typed* if type annotations are used in declarations of variables and operations. Object based programming languages encapsulate state and behavior in objects that are typed in most environments.

A number of so called *built-in* or *base types* are usually predefined by programming environments and more complex *user-defined types* can be built from more primitive ones using *type-constructors*. Types can be related because the same or similar operations can be applied to values of related types. For example an addition operation can be performed both for integer and floating-point numbers or any combination, a feature also know as *polymorphism*.

The most prominent relation between types, the *subtype* relation can be paraphrased as follows: type B is a subtype of type A if all operations of type A can also be applied to data of type B. The inverse relation of the subtype relation is the *supertype* relation. These relations are used in many forms within object-based language environments.

Type systems of programming languages differ and can be characterized through a number of criteria like strictness, homogeneity or flexibility. The techniques that can be used to overcome differences between type systems in the context of heterogeneous language migration vary and are discussed in the following sections.

strong / weak - static / dynamic - typed / untyped

Languages whose type system requires that the type of every variable, operation and expression can be verified with *type conformance rules* is called *strongly typed*. If the necessary checks can all be performed at compile time the language is said to be *statically typed.* Otherwise a language is said to be *weakly typed* or *dynamically typed* respectively.

Types are usually specified through keywords, so called *type annotations*, in the program text of languages. Type information that can be derived from other information is said to be typed implicitly. A language that uses type annotations is said to be *typed*, an *untyped* language does not use types annotations at all.

Strong static typing is usually favored because it allows to find common programming errors at compile time which then can be fixed early in the life cycle of applications. Dynamically typed languages check and catch some type errors at runtime. Untyped languages do not catch type errors at compile time but do usually employ special mechanisms for errors that arise during execution, like the “Message Not Understood” mechanism of Smalltalk [GoR1983].

The HLM migration mechanism requires type information from both the source and destination environments in order to identify the functionality that has to be transferred during migration. This determination is performed on the basis of interfaces, i.e. abstractions of language specific types that are represented in the GOAL language. The prototypical implementation of the mechanism therefore requires that both participating environments use explicit strong typing.

untyped (>) typed

If the necessary type information is not available from environments supposed to participate in migration it can be gathered through the implementation of *type inference* [PaS1994] for these particular environments using the enabling approach. Within source environments type information of objects to be migrated can be derived during the design of the respective applications through the use of appropriate development tools (see chapter 3 page 121).

typed (>) untyped

The type information of destination environments can also be computed prior to a migration request and object definitions that are added to an environment at runtime need to be processed upon their availability as well. The type information captured in interfaces transferred by the HLM migration mechanism to an untyped environment should be preserved to aid in future migrations. The transferred interface definitions are can be stored in files as GOAL source code for example.

The inference of types is usually a complex and time-consuming effort that is likely to be prohibitively long if attempted at the time of migration. Even if type inference can be applied prior to migration, the implementation of the type inference algorithm will imply an very high development effort. Migration between typed and untyped environments can therefore be characterized as prohibitively complex in both directions.

Differences between strong and weakly typed languages as well as between statically and dynamically typed languages will be of minor importance, as long as type annotations are available for all variables and signatures. In the context of the HLM migration mechanism only objects that are designed for migration, i.e. `OM_Migrateable` objects that use the defined standard interfaces can be migrated.

Whether type checking within the destination environment is performed statically or dynamically has to be considered during the generation and compilation of the native implementation during the transfer of semantics phase but has no direct implications for general migrateability of objects among the particular environments.

hybrid / pure

Object oriented languages can be either developed independently from scratch or implemented on top of existing programming languages. In the latter case language environments are called *hybrid* like C++ [Str1998] and CLOS [Ste1990], because they employ base types defined by the non object based host language. Object based languages that have been implemented independently are called *pure* like Smalltalk [GoR1983] or Self [Sun1992] as they usually employ only objects and do not have to distinguish between objects and base types[27].

The HML migration mechanism honors this distinction by defining standard interfaces for singular objects that may or may not be implemented through built-in types. Due to reasons of performance standard interfaces will be matched with predefined types in most environments during the generation of native code by the HLM migration mechanism.

hybrid (=>) pure

A migration of interfaces form a hybrid environment to a pure environment can be performed on the basis of the HLM migration mechanism unless the definitions of the base types, the standard interfaces are mapped to, have been changed within the destination environment. The compilation of the transferred functionality may fail or the behavior of the transferred objects may simply not be correct.

pure (=>) hybrid

If changes to base types have been applied in the context of the source environment a modified behavior of objects may result that can not be conveyed to the destination using the HLM migration mechanism. The standard interfaces of the HLM migration mechanism are considered to be static as they are implemented manually within the participating environments.

A alternative to static definitions of standard interfaces would be the requirements that standard interfaces have to be implemented via wrapper types around predefined types within all participating environments. The wrapper types can then be extended manually. Whether the necessary effort and the limited performance are worthwhile has to be examined by a standardization efforts of the HLM migration mechanism for its use in the real world.

parameterized types

Some languages define higher order types, so called *parameterized types* that allow the use of type variables within type declarations. Any defined type can be substituted for a type variable when the parameterized type is actually used. Parameterized types are only used by some strong typed languages like C++ [Str1991] or Eiffel [Mey1992] in order to avoid reimplementations of similar data-structures for each type they are used with. For example a list of type A may implement a list of integers as well as a list of strings.

---

[27] Such a distinction may exist on the implementation side for optimization purposes but not conceptually.

parameterless (=>) parameterized

The HLM migration mechanism is able to perform a migration from an environment without parameterized types to a environment with parameterized types if the destination environment excludes the parameterized types from the set of supported interfaces. The migrateability of objects of the source environment will depend on the number and kind of parameterless types defined by the destination environment.

If a migration from an parameterless environment to an environment with parameterized types is not successful, an additional negotiation phase can be implemented as an extension of the HLM migration mechanism. The second negotiation phase can attempt to find appropriate bindings for unsupported interfaces of the source through the determination of appropriate bindings for parameterized types of the destination. However, the corresponding interface definitions will have to be changed accordingly using adaptive migration.

parameterized (=>) parameterless

The reverse migration from an environment that supports parameterized types to an environment with no support can only work if all parameterized types can be converted to normal types prior to the negotiation phase. The bindings of type variables are used for the objects to be migrated have to be determined and the corresponding interface definitions have to be generated during the development phase.

The alternative change of the GOAL language to support parameterized types will lead to a partitioning of migrateability as only few environments support parameterized types. Migration of object that use parameterized types will then be possible between environments that implement parameterized types but not with environments that do not.

4.4.2 Object-Orientation and Prototypes

Programming languages that operate on objects can be characterized in several ways. Following the systematic of Wegener [Weg1990] encapsulation and message passing can be defined as the most general characteristics of all so called *object-based* languages, i.e. languages that support the concept of objects.

Object based languages can be further subdivided into *prototype based* and *object-oriented* languages according to their use of the concepts of prototypes, cloning and delegation or classes, instantiation and inheritance respectively. Figure 4.n illustrates the relationship between the different kinds of languages.

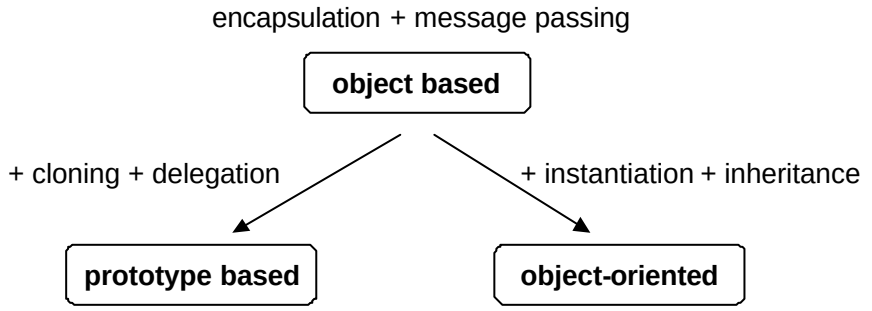

Figure 4.n: The relation between object-based, prototype-based and object-oriented language environments is characterized by the extension of the fundamental concepts of encapsulation and message passing through either cloning and delegation or instantiation and inheritance respectively.

As the concepts of object-based language are all interdependent, brief overview of the individual concepts of object-oriented and prototype-based language environments is given in the following and a more detailed discussion on the basis of more general categories is provided in the subsequent subchapters.

object-oriented

Object-oriented environments define the structure and behavior of objects through classes that can themselves be represented as objects as for example in Smalltalk [GoR1983] or that may only exist as a textual representations within corresponding source files like in C++ [Str1991]. *Instantiation* is the process of creating a new object as defined by a class. An object instantiated from a class is called an *instance* of that class. The *instance of* relationship connects an object with its class and the subtype relationship between classes is called *inheritance*.

The subtype relationships can be a hierarchy in the case of *single inheritance* or a directed acyclic graph as in the case of *multiple inheritance.* I.e. each class can be a *subclass* of one or more *superclasses*. The access to the state of objects can be protected in terms of the subtype relationship and the method lookup can differ whether an instance or a class object is addressed by a message passing expression. Figure 4.o illustrates the concepts of object-oriented language environments.

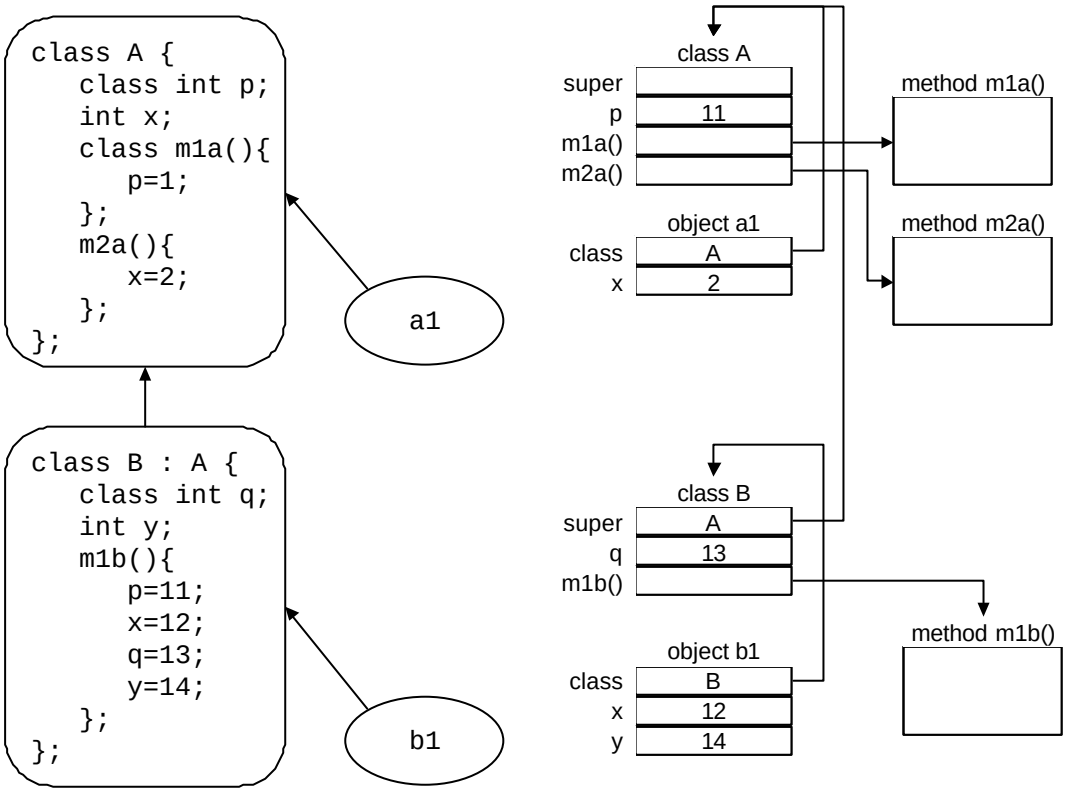

Figure 4.o: Some object-oriented languages distinguish between class- and instance variables and -methods. In the above example class A (shown here as a rounded rectangle) defines a class variable p, an instance variable x as well as a class method `m1a()` and an instance method `m2a()`. Class B defines an instance variable y and an instance method `n1b()`. The example depicts the situation after two instances a1 and b1 have been created for class A and B respectively and the method `m1a()`, `m2a()`, and `m1b()` have been invoked for class A, instance a1 and instance b1 respectively. The corresponding in-memory representation is shown schematically on the right side.

Message passing is used to trigger the invocation of methods that are determined in a process called *method lookup*, *dynamic dispatch* or *dynamic binding*. The method determined is executed in a context that includes the object the message was originally send to as the receiver of the message regardless of the class or superclass the method invoked is actually defined for. Most object-oriented languages invoke methods in a synchronous manner where the sender of a message waits for the return of the result.

prototype-based

Prototype based environments employ the concept of prototypes, cloning and delegation in addition to encapsulation and message passing. *Prototypes* are normal objects that are self-describing and can be used to create other objects through a process called cloning. *Cloning* creates a copy of an object and establishes a so called *parent* reference that connects the newly created object and its prototype.

The parent reference establishes a partial subtype relationship between prototypes that is called *delegation* and is used to forward variable accesses and messages that can not be handled by an object itself. Only variables and methods that are defined or redefined for the object respectively are handled by the object directly. Figure 4.p illustrates the concepts of prototype-based language environments.

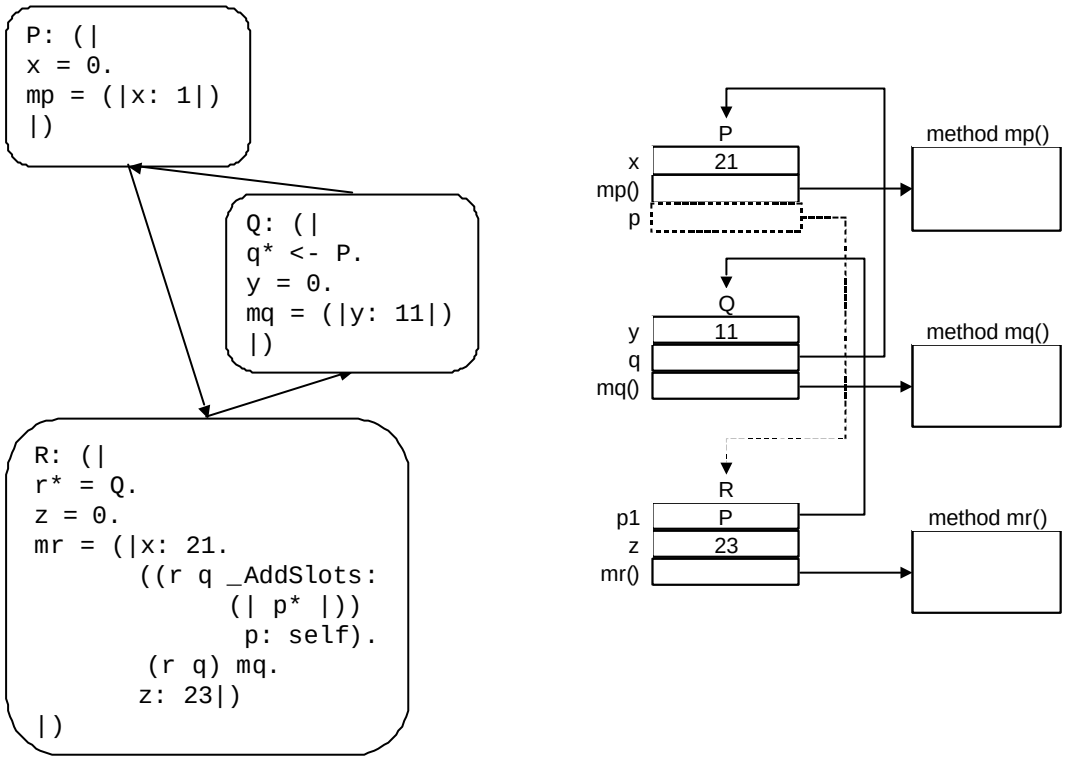

Figure 4.p: Some prototype based languages determine delegation between prototypes for each variable and method separately as for example Self[28]. In the above example prototype P defines variable x and method `mp()`, prototype Q defines a variable q with P as a parent, variable y and method mq, prototype R defines a variable r with Q as a parent, variable z, and method mr. The example depicts the situation after method mr has been executed for prototype R as follows. The value 21 is assigned to variable x, which is found through the parent relationships of R in prototype P. A variable p with R (the value of self, the pseudo variable for the receiver of the message) as a parent is added to prototype P, which is accessed through the variables r and q Method mq, which is found through the parent relationship as being defined by Q, is invoked for variable p, which is found through the parent relationship of R in prototype P. Method mq assigns the value 11 to variable y, which is found through the parent relationship of P in prototype Q. The corresponding in-memory representation is shown schematically on the right side. (The variable and parent relationship that is added through mr is shown here with dashed lines).

[28] The example shows simplified prototype definitions of the Self language [Sun1992].

A single or multiple parent relationships can be defined for prototypes up to the point where each variable or method of a prototype can be delegated to a different prototype individually including cyclic parent relationships. In some prototype languages variable and method can be protected from access and invocation by objects other the individual prototype itself.

The access to state as well as the method lookup honor the diverse parent relationships and determine each variable to access and each method to invoke anew, based on the individual receiver of the corresponding message. The invocation of methods is performed in a context that usually included the original receiver of the message but may also include the object the variable or method is defined for as the object the message has been forwarded to. The execution of message passing expression can be performed as synchronous as well as asynchronous invocations.

object-based Migration

The HLM migration mechanism deliberately abstracts from classes and prototypes through the use of interface definitions that do not have to be implemented as objects but can be managed on the basis of source code files. Each environment can map interface definitions to individual abstractions as appropriate. The concept of single inheritance and synchronous message passing are also used by the HLM migration mechanism as a common ground among all object based language environments.

The following sections discuss various sets of related concepts of object-based languages as well as the techniques that can be used to enhance or extend the HLM migration mechanism in order to support them. The concept of inheritance and delegation, instantiation and cloning, various forms of access to state as well as several method lookup schemes are discussed. Different forms of message passing are discussed in a subsequent subchapter.

Single Inheritance / Multiple Inheritance / Delegation

Object-based environments offer different forms of inheritance or delegation to share state and behavior among objects. Various forms like single or multiple inheritance as well as delegation can be determined among existing language environments. Some environments offer combinations of inheritance and delegation, as for example Calico [Bel1993], Java [GJS1996] or Objective C [Cox1992].

Single Inheritance

In an environment that supports single inheritance a definition of an object can only depend on one other object definition in order to specify the state and behavior of objects. As a result, all object definitions form a hierarchy and the determination of variable and method definitions can be performed unambiguously. An example of single inheritance among object definitions is shown in figure 4.o above (see page 193).

Individual object definitions can redefine variables or methods that are otherwise inherited from their ancestors and new variable and method definitions can be added. The differentiation of descendants of a object definition is also called *specialization*. Few language environments even allow the deletion of inherited features, that will not be available for all descendants of the object definition that defines the deletion.

Multiple Inheritance

Object-based languages that support multiple inheritance allow several ancestor for each object definition. As a consequence the subtype relationship is a directed acyclic graph and the determination of variable and method definitions can be ambiguous, as a given feature can be inherited along different paths of the inheritance lattice.

Various solutions to the problem of ambiguity have been proposed and are used within language implementations. These range from special hints for ambiguous cases through compiler checks preventing ambiguity to a total ordering of interfaces. Figure 4.q illustrates multiple inheritance.

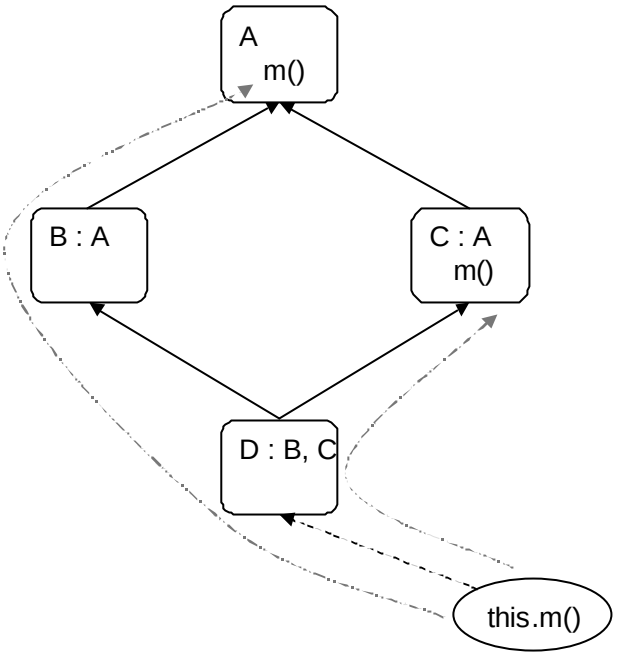

Figure 4.q: Multiple inheritance can lead to ambiguities. In the above example The object definition D (shown here as a rounded rectangle) inherits from B and C while B and D themselves both inherit from A. The determination of the definition of method `m()` for an instance (shown as an ellipses) may lead to the definition of A or C depending on the particular rule chosen for the determination (the different path of inheritance are shown as broken gray lines).

Multiple inheritance allows to inherit different features from different ancestors and is perceived to simplify the design of object-oriented applications in some cases. Alternative techniques that offer similar capabilities for single inheritance have been developed though. Whether multiple inheritance offers significant advantages to single inheritance that justify the additional effort of dealing with ambiguity is an open problem.

Delegation

Prototype based environments use the concept of delegation among objects that is related to inheritance among object definitions. The parent relationship between objects as established by cloning resembles single inheritance between object definitions but some environments allow multiple parents to be defined. In the most complex case arbitrary delegation of each individual variable or method to a different object definition will be possible.

The determination of the definitions of variables or methods is performed on a case by case basis following the parent relationships except for variables and methods that are defined for he particular object directly. As delegation relationships can become arbitrarily complex, including cyclic structures different forms of prevention and resolution of ambiguities have been developed. An example of delegation is shown in figure 4.p above (see page 194).

The HLM migration mechanism requires a restriction to the use of single inheritance among the participating environments. Only interfaces that use single inheritance either from the source or destination environment can be considered during the negotiation process in order to ensure compatibility among environments.

single inheritance (=>) multiple inheritance

A migration form an environment that supports single inheritance to an environment that supports multiple inheritance can be performed using the HLM migration mechanism. The determination of the supported interfaces within destination environment has to be restricted to object definitions that use single inheritance as multiple inheritance can not be considered by the negotiation algorithm.

multiple inheritance (>) single inheritance

A migration form an environment with multiple inheritance to an environment with single inheritance is not possible in the general case. Only the special case, that the multiple inheritance lattice of the dependent interfaces of the source environment can be matched with a single inheritance hierarchy of the destination environment can be supported in the context of the HLM migration mechanism.

Informed migration will have to be used as the interface definitions and object representations to transfer will have to be generated on the fly, depending on the available interfaces at the destination. The symmetric and transitive nature of migration may be lost as the migrated objects will be adapted for the destination environment specifically.

multiple inheritance1 (>) multiple inheritance2

The same holds for migration between environments with different multiple inheritance schemes. In the general case a match of different inheritance schemes is not possible due to the computability problem. Even in cases where both environments employ the same technique to avoid ambiguities the transfer of objects definition may still be impossible due to contradictions between the inheritance lattices.

In order to support cases where different multiple inheritance scheme match, the HLM mechanism can be extended to perform the necessary adaptation of the object definitions to be transferred. The determination whether two multiple inheritance scheme are compatible has to be done manually prior to migration.

The apparently simple alternative, the addition of multiple inheritance to the GOAL language would implies a commitment to a certain kind of ambiguity prevention that would rule out all other forms. Likewise environments that support only single inheritance could no longer participate in migration based on the HLM migration mechanism.

single inheritance (=>) delegation

Migration from environments that support single inheritance to environments that support delegation can be performed by the HLM migration mechanism provided that the destination environment supports a single inheritance compatible form of delegation and that the determination of supported interfaces is also based on single inheritance.

Apart from the necessary management of interface definition within the destination environment no changes to the HLM migration mechanism will be necessary. The code generation for the native object implementations of the destination environment can take care of the necessary conversion of object definitions.

delegation (>) single inheritance

The general case of migration from environments that support delegation to environments that support single inheritance is not possible except for objects that are confined to delegation in a single inheritance style. As single inheritance is covered by delegation, no provisions for the determination of supported interfaces are required.

A match of arbitrary complex delegation schemes with a single inheritance hierarchy is not possible in the general case, but whether a subset of prototypes of a delegation based source environments are defined in a simple inheritance style can be tested automatically. Such a test can be included in the check phase of the implementation of the HLM migration mechanism for the prototype based environment.

multiple inheritance (>) delegation

Migrations from environments with multiple inheritance to environments with delegation can only be performed if the multiple inheritance lattice of the source can be matched with the delegation scheme at the destination. Such a determination has to be performed manually prior to migration due to the problem of computability.

If the compatibility is determined for a combination of environments the HLM migration mechanism can be extended to perform the necessary adaptations of the interface definitions

using informed migration. Multiple inheritance will have to be added to the GOAL language in order to be able to perform the negotiation.

delegation (>) multiple inheritance

Migration from environments with arbitrary delegation to environments with multiple inheritance will not be possible in the general case due to the problem of computability. If the source environment uses delegation in a multiple inheritance style, migration will be possible, but such a determination has to be made manually.

If two environments can be determined to be compatible the HLM migration mechanism can be extended to perform the necessary adaptation of object definitions using informed migration. A version of the GOAL language that supports multiple inheritance has to be used in order to negotiate the supported interfaces and to transfer the necessary interface definitions.

delegation1 (>) delegation2

Migration between environments with different delegation schemes will also not be possible in the general case as a match of arbitrary delegation schemes can not be computed. The delegation mechanism of two prototype-based environments can match nevertheless in special cases and the HLM migration mechanism can be extended to perform the necessary adaptations accordingly.

The alternative addition of delegation to the GOAL language would require a commitment to a fixed style of delegation that precluded the migration with environments that are not compatible with this style. Such an addition would also not be sufficient for migrations between prototype-based and object-oriented environments.

State Access

Object-oriented languages that implement classes as objects often distinguish between class- and instance-variables as well as class- and instance-methods, for example Smalltalk [GoR1983]. While class-variables exist only once within the corresponding class object and are shared among all instances of the particular class, instance-variables are created with each instance individually and are accessed and changed separately.

Class methods are distinguished from instance methods as class methods can only be invoked for the particular class object, while instance methods can be invoked for all instances of the class. Some languages employ constructs that enable the access to variables of objects independent from the message passing mechanism like `friends` in C++ [Str1991].

Some languages allow direct access to variables only from methods defined as part of the corresponding class-definition and otherwise require so called accessor-methods like CLOS [Pae1993]. Object-oriented languages define various levels of protection for variables and methods that enable or prohibit access to and invocation of state and behavior.

Prototype base languages do also distinguish different forms of access to the local state of objects and share state through delegation directly. An object that is cloned from a prototype delegates access to local variables and the invocation of methods to its prototype unless variables or methods are assigned to or redefined for the object itself. Cloned objects effectively share the variables and methods of their prototypes until specialization.

The HLM migration mechanism ensures the compatibility between different forms of encapsulation through the requirement that local variables of objects can only be access from local methods. Only message passing can be used to transfer state between objects and appropriate accessor signatures have to defined. The access to the state of objects is essentially subsumed under the message passing mechanism.

not shared (=>) shared

A migration from an language that does not support sharing of state among objects to an environment that does can be performed by the HLM migration mechanism provided that the supported interfaces do not rely on sharing of components. The interface definitions that are transferred do not make use of sharing and can be compiled into native implementations.

The determination of the supported interfaces can leave the shared components out of the interface declarations that are transferred to the source environment. The implications are that less interfaces are available for negotiation but the shared declarations could not have been matches anyway.

The alternative to add shared state to the GOAL language would require a commitment to one kind of sharing and would rule out all other kinds. Furthermore the use of declarations of shared state would imply additional dependencies between object that have to be considered during the check and negotiation phases [Ben1990, PhZ1997].

shared (>) not shared

A migration from an environment that supports shared state to an environment that does not will not be possible without changes to the interface definitions transferred. A sharing of state that depends on the type of the objects within the source environment can not be expressed within the destination environment but can be simulated using appropriate generated code.

The state that is shared within source environments in an automated fashion can be substituted with the definition of appropriate accessor signatures and the use of common objects that manage the shared state. The interface definitions for the common objects have to be generated at migration time and the changes to the definitions of existing objects have to be performed through adaptive migration by the HLM migration mechanism.

shared1 (>) shared2

A migration between two environments that support different forms of sharing will not be possible in the general case. If the sharing model of the source can be subsumed under the sharing model of the destination migration will be possible through adaptive behavior of the HLM migration mechanisms.

If the destination offers less sharing than the source environment migration will only be possible with techniques described in the previous case that require significant changes to the definitions of the objects involved and the appropriate extensions to the generation of the object definitions to be transferred by the HLM migration mechanism in the form of adaptive migration.

unprotected => protected

A migration form an environment that does not protect state to an environment that does protect state can be performed by the HLM migration mechanism. The interface definition transferred will not have to be changed as the objects of the source environment to not make use of protective features.

The determination of the supported interfaces will have to leave out declarations of protected state that will not be used to the migrated objects anyway. The negotiation algorithm is not affected and the native implementations of the migrated objects can be compiled as the existing object definitions that use protection are not changed.

The alternative to add declarations of protection of state to the GOAL language would require a commitment to one form of protection that would rule out all others. The use of protection by objects to be migrated would also rule out their migration to environments that doe not support protection of state.

protected (=>) unprotected

A migration from a language that offers protection of state to an environment that does not will be possible in the context of the HLM migration mechanism if all protected components of the source can become unprotected in the destination environment. This simple change can be performed during the generation of interface definitions at the source.

Unfortunately, the “out-coming” of the protected state of objects can have devastating effects if the interfaces definitions transferred are used in manual development efforts within the destination. Furthermore the change of protection of the migrated objects will inhibit symmetric migration of the object back to their source environments without further provisions.

protected1 (>) protected2

A migration between environments with different protection models will not be possible in the general case. In the special case that the protection model of the source can be subsumed under the protection model of the destination environment, migration will be possible in the context of the HLM migration mechanism provided the appropriate adaptations can be applied.

Method Lookup and Invocation

Object based environments use message passing between objects in the form of message passing expressions to execute the underlying semantics. Regardless whether a message is actually communicated between two objects or whether a message is only used as an abstraction, a message passing expression is executed in two steps.

When a message is received by an object a so called *method lookup* or *dynamic dispatch* is performed that determines the actual method to be executed. When a match is found a so called *method invocation* is performed that uses *dynamic binding* to construct an activation record that contains the receiver and the parameters of the message as local variables. The code of the method is executed in the so called *message context* defined by the established activation record.

Object-based environments differ in the way they perform both the method lookup as well as the method invocation. Although similarities can be found among most environments or can at least be achieved through appropriate restrictions some particularities can prevent the migration between individual environments altogether.

Method Lookup

The method lookup that is performed by object-based languages depends primarily on the input factors that are used to guide the lookup which are either contained in the message that was sent or are established by the runtime environment. The method to invoke is usually determined on the basis of the receiver of the message, the *selector* or name of the message and the parameters of the message. In some cases the type of the result as well as the sender of the message are also considered.

Even environments that use the same input factors for their method lookup algorithms may differ in the way these factors are interpreted. For example some languages interpret null values as implicit subtype of any type and allow null values as parameters. Other language are unable to cope with “typeless” null values as parameters.

From a theoretical point of view the dependencies between parameters of a message and its result can be characterized as covariant or contravariant. *Covariance* and *contravariance* respectively specify whether methods arguments vary with the subtype relationships of the objects they are defined for or not. As a result covariant and contravariant method lookup are incompatible in most cases.

Fortunately, only very few languages use covariance, for example Eiffel [Mey1992], and most object-based languages use contravariance. The theoretical differentiation of covariance and contravariance does not matter in practical situations due to other preconditions of the HLM migration mechanism (see also page 211).

The method lookup algorithms take the dependencies between the object definitions into account if an appropriate method is not found within the definition of the receiver object. The method invoked depends on the use of single or multiple inheritance or delegation relationships that exist between the relevant object definitions.

Most environments ensure at compile time through type checking that each message passing expression can be executed, i.e. a matching method can be found. Some languages, especially untyped languages do not ensure that a matching method exists and raise an error if no matching method can be found for a message expression at runtime. Smalltalk [GoR1983] for example sends the `MessageNotUnderstood` if no matching method can be found.

Some languages provide a construct for the forwarding of messages between objects. The `super` keyword is often used to forward a message to the ancestor of a receiver if it can not be

handled by a matching method of the receiver directly. Some prototype based languages offer a `resend` construct that can be used to forward the message to an arbitrary object.

As an extreme case the Common Lisp Object System (CLOS) implements a concept called *method combination* that allows the execution of a number of methods in reaction to the receipt of a single message. The additional methods that are executed `before` and `after` the main method can be defined throughout the multiple inheritance lattice of CLOS that is totally ordered.

CLOS also implements methods as generic functions, a concept that is found in the form of Multimethods also in other object-based languages like Cecil [Cha1993]. *Generic functions* or *multimethods* generalize the concept of methods as a generic function does not belong to a specific receiver type but can be invoked for a number of receiver types.

The method lookup algorithms that is used in the context of generic functions or multimethods determines the method to execute not only on the basis of the receiver type and the selector of the message but takes also all other parameters of the message into consideration. The method to execute is identified as the most specific one with regard to all parameters of the message passing expression.

method lookup1 (=>) method lookup2

In the context of the HLM migration mechanism the definitions of methods are confined to the most simple case. Methods have to be distinguished by the type of the receiver and of the selector of a message alone and the use of null values has to be avoided within message passing expressions.

As a consequence the designer of an application that uses the HLM migration mechanism has to make sure that all methods are defined appropriately and that all variables that are used as parameters of messages are initialized before they are used. Appropriate development tools can be used to ensure these preconditions of migration (see chapter 3 page 121).

Support for more elaborate method lookup schemes is not possible in the general case due to the problem of computability. Whether different method lookup mechanisms can be matched and how one form of message passing can be converted in to a matching different one can only be determined manually.

If a match of the method lookup algorithms between two environments is identified the HLM migration mechanism can be extended to perform the appropriate adaptations of the object definitions to be transferred. The GOAL language has to be changes appropriately and the determination of the supported interfaces take the compatibility of signatures into consideration.

Method Invocation

The method invocations that are performed among environments can also differ as the bindings of implicit parameters that are establish prior to the execution of the method code can deviate. The receiver of a message is usually bound to a special variable called `self` or `this` that is part of the invocation context.

Apart from the parameters of the message passing expressions further implicit parameters can be bound like the sender of the message or the object a message has been forwarded to. Further but very uncommon alternatives of message passing, e.g. the use of a receiver of the result of a method invocation that is independent of the receiver of the message, are not discussed here.

Some language environment also distinguish whether parameters of message passing expressions are passed by value or reference, i.e. whether parameters are used as read only input or can be changed as writeable in/out or out parameters. The use of call-by-reference requires the availability of pointers within language environments (see page 209).

invocation1 (=>) invocation2

The GOAL language uses only the `this` keyword as an implicit parameter of a method invocation that is bound to the receiver of a message. The HLM migration mechanism requires

to confine the design of objects to be migrated to the use of no other implicit parameters, even if they are available within source environments. Additionally all parameters are considered to be passed by value even if the object implementations within destination environments differ, i.e. the writeable “out” character of parameters that may exists is never used.

Migration between environments with different invocation contexts can not be performed in the general case as implicit variables that are available within one environment and that are used by the objects to be transferred can not be reconstructed in a destination environment without changes to the runtime environment itself.

Instantiation / Cloning

The creation of objects differs among object-based environments. Object-oriented languages employ a process called *instantiation* that creates a new object on the basis of its definition that is represented by a class object. The instance variables defined for the class object are allocated and a instance-of relationship to the class object is established.

Object-oriented languages vary in the ways the initialize the instance variables of newly created objects. Some define so called *constructors*, i.e. methods that are invoked implicitly upon the creation of objects. Other rely the explicit initialization of instance variables through the user. Likewise some languages define *destructors*, i.e. methods that are invoked implicitly when an object is destroyed (see also page 209).

Prototype based environment use a process called *cloning* that in the simplest case creates a new object from a prototype as a clone that contains no more then a parent relationships to its prototype. Only variables and methods that are defined directly for the new objects are added to its internal representation.

instantiation (=>) cloning

Migration from environments that support instantiation to environments that support cloning can be performed in the context of the HLM migration mechanism through adaptive migration. The translation of the transferred interface definition has to be done in such a way that all variables that are considered instance variables are redefined for the cloned object within the destination environment.

The determination of supported interfaces within the destination environment has to be confined to predefined interface definitions though. If variables of prototypes can be redefined at runtime, the set of variables that are redefined for a particular prototype can not be determined due to the problem of computability.

cloning (=>) instantiation

Migration from environments that support cloning to environments that support instantiation if redefinitions of cloned objects are restricted to be performed during the creation of objects. In such a case the object definitions of prototypes can be matched with supported interfaces and the local variables will be allocated through the instantiation process automatically.

The alternative would require reflective capabilities within the destination environment as the definitions of objects of the destination environment would have to be changed at runtime if redefinitions of objects at the source have to be matched with equivalent operations within the destination environment.

instantiation1 (=>) instantiation2

Migration between environments that offer different forms of instantiation will only be possible if the allocation of memory for each newly created object as well as the initialization of instance variables is performed in a compatible way. The allocation of instance variables is performed similarly among most object-oriented languages, i.e. instance variables of the objects definition as well as of its ancestors are allocated collectively. Unfortunately, the initialization of newly created objects differs among languages.

In order to avoid problems with different initialization mechanisms of the HLM migration mechanism requires that variables of newly created objects are either initialized with null values

automatically through the appropriate means or that explicit initialization is performed through a dedicated signature like `initialize()`.

Object-oriented / Prototype-Based

Despite the many differences that exists between object-oriented and prototype-based environments, migration between such environments can be performed in the context of the HLM migration mechanism if the appropriate restrictions are applied to the design of the objects to be migrated in either direction.

classes (=>) prototypes

The HLM migration mechanism can be enhanced to support migration from object-oriented to prototype based environments although only prototypes that are designed for the migration mechanism can be used as supported interfaces. E.g. delegation has to be confined in a single inheritance style, redefinitions have to be static, and implicit invocation parameters can not be used.

The implementation of the interface set can be performed straightforwardly as the single inheritance relationship that can be resembled by the parent relationships. Instantiation can be resembled by cloning because several "instance" objects can be cloned from a single prototype that fulfills the role of a "class" object, i.e. instance variables have to be assigned to each cloned object individually. The translation of the transferred interface definition into native source code has to be performed accordingly.

prototypes (=>) classes

A migration from a prototype based environment to a object-oriented one will be possible in the context of the HLM migration mechanism if the objects to be migrated use delegation only in a single inheritance style, redefinitions can be determined statically, state is not shared and implicit invocation parameters are not used. As each prototype will define its own interface a new class will have to be created within the destination environment for all prototypes that do not share interface definitions.

The generation of interface definitions in the GOAL language has to translate the prototype style syntax to a class style syntax, e.g. cloning expressions have to be expressed as object creation expressions. A check of the conditions for the migration of prototypes has to be performed statically at design time through appropriate development tools and can be combined with the generation of interface definitions.

Migrations between class-based and prototype-based environments will not be possible in the general case as prototype based systems are generally more flexible than object-oriented ones. Arbitrary delegation the redefinition of variables and methods at runtime, the different sharing and protection of state as well as different implicit invocation parameters will inhibit migration in the general case.

4.4.3 Message Passing

Object based languages define the behavior of objects through methods that can be invoked through message passing. Messages that are sent between objects control the flow of execution in object based systems. When a message is received by an object, an appropriate method is determined and invoked using the execution context specified by the message.

The execution of messages passed between objects can be performed synchronously or asynchronously. With *synchronous message passing* an object that sends a message to another object will wait for the result of the method called to be returned. The objects that are subject to synchronous message passing are called *passive objects* as they do not employ they only react to messages that are sent to them but are not able to conduct computations on their own, independent of message passing. Figure 4.r illustrates synchronous message passing.

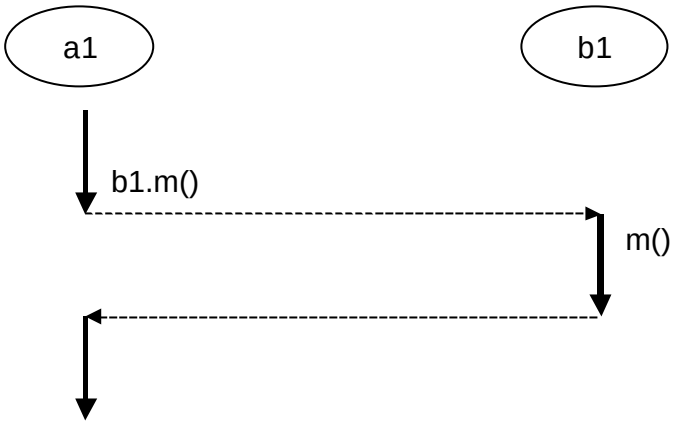

Figure 4.r: During synchronous message passing the sender object blocks until the receiver answers. In the above example object a1 invokes method `m()` of object b1 and waits for b1 to return the result of its computation (computations are shown here as thick arrows while communications are shown as thin dashed arrows).

Alternatively *asynchronous message passing* can be used where the object that sends a message does not wait for the result to return but continues with its own independent computation. The result computed by the receiver of the message will be returned in the same way as a result message, i.e. through a message to the sender of the original message.

Objects that implement their own computations are called *active objects*. Active objects manage incoming messages via so called *message queues* that store incoming messages until the are processed through method invocations on a first come first serve basis. Figure 4.s illustrates asynchronous message passing.

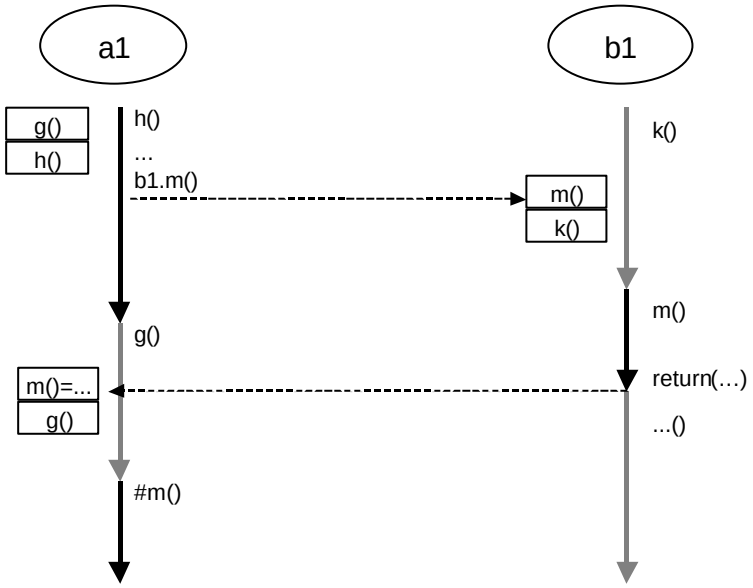

Figure 4.s: With asynchronous message passing between objects (shown here as ellipses) messages are buffered in message queues (shown as rectangles), processed on a first come first served basis and their results are sent back as messages as well. In the above example object a1 sends a message `m()` to object b1. The message is queued until the previous message `k()` is processed. The result of the invoked method `m()` is send back to a1 and is also queued there until the intermediately invoked method `g()` is processed. (The sequence of computations that are related to the message `m()` are shown here as thick arrows, other computations are shown as gray arrows, communications between the objects are shown as dashed arrows)

Active objects act much like operating system processes [Hoa1978] that communicate with each other through inter process communication mechanisms. Asynchronous message passing is also similar to the communication between distributed systems that operate independently while remote method invocation resembles synchronous message passing in the presence of distribution. Unlike their operating system relatives active objects usually share a single address space and require a runtime environment that is capable of executing multiple threads.

Synchronous and asynchronous message passing as described above mark two extremes of a spectrum of implementation choices that consists of all kinds of multi-threaded environments in between these two extremes. With asynchronous message massing and active objects each object employs its own thread, but no threads exists outside of these active objects. With synchronous message passing only a single thread exists within the whole environment.

Multi-threaded environments, i.e. environments that enable the execution of multiple flows of control in parallel, can implement synchronous message passing as well as asynchronous message passing. Different means of synchronization between threads like semaphores or monitors are used to coordinate the access of several threads to shared resources. Table 4.a shows the relation between form of threading and of message passing.

	Single-Threading	**Multi-Threading**	**Active Objects**
Synchronous Message Passing	x	x	-
Asynchronous Message Passing	-	x	x

Table 4.a: The possible combinations of synchronous and asynchronous message passing with single- or multithreading as well as active objects respectively.

The containment of threads within active objects can be relaxed in multi-threaded environment such that both objects that encapsulate threads as well as passive objects can exist side by side. Several flows of control can be executed in parallel within these environments that involve method invocations of different passive objects including synchronized invocations as well as rendezvous with encapsulated threads.

Additionally multi-threaded environments may implement objects that encapsulate several threads as well as several threads that execute only among many passive objects. Which form of multithreading and combination of passive and active objects can be implemented by a particular environment depends on the built-in primitives that re used to manage the threads.

The HLM migration mechanism relies upon synchronous message passing and does not support the transfer of computations. As mentioned previously the HLM migration mechanism can be extended to support heterogeneous migration of computations if the participating language environments are enabled accordingly. Migration of computations can also be supported in multithreaded environments if the necessary synchronization is performed (see also page 179).

synchronous (=>) asynchronous

A migration form a synchronous to an asynchronous environment can be performed in the context of the HLM migration mechanism if the interface definitions transferred are translated in such a way that asynchronous message sends will wait for the return of the result. However the asynchronous nature of the objects if the destination environment will have to be considered in the determination of the supported interfaces.

asynchronous (>) synchronous

A migration from an asynchronous to a synchronous environment will only be possible if the destination environment supports multithreading and a new thread can be started for each incoming message. The GOAL language will have to be change to differentiate between

synchronous and asynchronous message sends and the necessary code has to be generated in the context of the HLM migration mechanism using adaptive migration.

single-threaded (=>) multi-threaded

A migration from a single-threaded environment to a multi-threaded environment can be performed in the context of the HLM migration mechanism if a new thread can be created for the transferred objects. The determination of the supported interfaces has to take the multi-threaded nature of some object definitions of the destination into account.

multi-threaded (>) single-threaded

A migration form a multi-threaded environment to a single-threaded environment will not be possible in the general case as the parallel nature of objects of the source can not be recreated within the destination. Migration will be possible only in the special case where passive objects that employ not more then a single thread are migrated from the source to the destination environment. The later case is assumed in the context of the HLM migration mechanism.

Multi-threaded1 (>) multi-threaded2

Whether migration can be performed between environments that employ different forms of multi-threading depends on the individual case. Especially the extreme form of multi-threading that implements only active objects is problematic in this regard. Environments that employ only active objects can not recreate combinations of active and passive objects or multiple threads among passive objects alone. Active objects can however be migrated to multi-threaded environments if threads can be encapsulated by objects within the destination environments.

4.4.4 Other Concepts

The following concepts of object based languages are not available within all object-based environments but they are prominent enough to deserve a detailed discussion. Although a partitioning of migrateability will result from the support of these concepts possible techniques to augment the HLM migration mechanism as well as general aspects of migration in the context of these concepts are discussed nevertheless.

Only the most interesting concepts are chosen here, namely reflection and introspection, blocks, exceptions, as well as pointers, garbage collection and finalization. A great number of further concepts exists but not every concept of all existing object-based language environments can be covered here for reasons of space.

reflection

The term *reflection* denotes a concept implemented by some object-oriented languages like Smalltalk [GoR1983] or CLOS [Pae1993] that renders the state and behavior of objects accessible within object-oriented programs through the definition of the higher-level objects that are used to define instance- and class-objects. A similar concept called *introspection* has been defined for prototype based environments like Self [Sun1992] and a read-only form of reflection is available within some languages like C++ [CKW1998] under the name *runtime type information*.

Within object-oriented languages reflection is often implemented through the threefold instance - class - metaclass hierarchy in the tradition of Smalltalk [GoR1983]. As the behavior of instance-objects is specified by class-objects the behavior of class-objects is defined by metaclass-objects. The behavior of metaclass-objects is usually not defined by another level of abstraction but the potentially unlimited instance-of hierarchy is limited by a predefined class-object called MetaClass that defines the common behavior of all metaclasses[29] including itself.

With reflection the definitions of state and behavior of objects are available at runtime in the form of class or metaclass objects and can be changed more or less directly. For example, new variables can be added to the definition of objects or methods can be redefined or deleted.

[29] As the root of the instance-of tree the metclass Metaclass defines the only exception to the irreflexive nature of the instance-of relationship.

Whether these changes are propagated to the existing objects or just valid for new created objects depends on the actual implementation.

Some prototype based systems employ *introspection,* a concept similar to reflection that makes the definition of state and behavior of prototypes directly accessible. Some prototype languages like Self [Sun1992] use so-called *mirror* objects that can be used to manipulate the structure and behavior of prototypes. Figure 4.t shows the Instance - Class - Metaclass hierarchy.

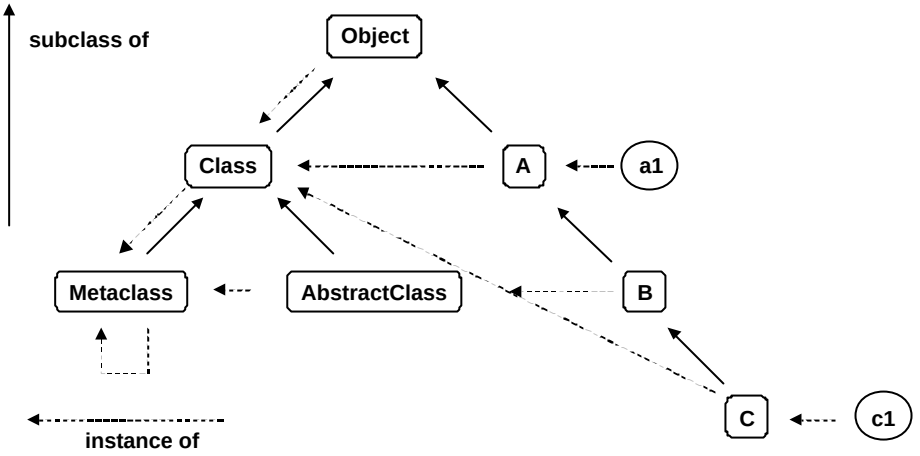

Figure 4.t: The Instance - Class - Metaclass hierarchy is used by object-oriented environments to implement reflection. In the above example the "subclass-of" relationship is shown from bottom to top and the "instance-of" relationship is shown from right to left.

The HLM migration mechanism does neither support nor rely upon reflection or introspection. A implementation of the mechanism may benefit from reflective or introspective features of a language environment because information about interfaces of objects can be gathered at runtime. Especially the advanced forms of migration like adaptive or informed migration will benefit from the availability of reflective or introspective features as only the necessary details of interfaces in question have to be retrieved.

non-reflective (=>) reflective

A migration from an environment that does not offer reflection to an environment that offers reflection can be performed in the context of the HLM migration mechanism as the object to be transferred will not employ any reflective capabilities. The determination of the supported interfaces has to filter out any reflective interfaces that can not be represented in the GOAL language anyway.

reflective (>) non-reflective

A migration from a environment that offers reflection to an environment that does not support reflection will not be possible without significant changes to the destination environment. Since reflective behavior can not be recreated without the necessary built-in functionality the definition of the particular language or the implementation of the environment would have to be changed.

reflective1 (>) reflective2

A migration between two environments that support different forms of reflection will not be possible in the general case. A determination whether two forms of reflection can be match will not be possible due to the problem of computability. If two environments can be matched in this regard manually the GOAL language has to be changes accordingly and the HLM migration mechanism can be extended to perform the necessary adaptations.

blocks

Most object based languages employ a set of common control structures like conditions (`if then else endif`), multiple-conditions (`switch case default endswitch`), or loops (`while do endwhile`, `repeat until`, `do while`, or `for to do endfor`). Unconditional

jumps like `goto` with labels are not common but various forms of non local return are used that are used as well (see also page 208).

Some languages like Smalltalk [GoR1983] extend the notion of objects to control structures and use the concept of *blocks* as a means to defer the evaluation of statements independent of their lexical scope. Blocks are not evaluated at their lexical position but only when they are sent an `eval` message. Blocks can therefore be used to construct arbitrary user defined control structures.

The HLM migration mechanism uses a set of common control structures that is found in most object based as well as procedural languages. Blocks are deliberately not supported by the HLM migration mechanism as only few object-based language environment support the block concept.

or alternatively appropriate conversions can be made through adaptive migration if a mapping can be found between the control structures of the source and the constructs of the destination environment prior to migration.

Such a mapping can not be determined automatically due to the halting problem in the general case if for example blocks are used. Migration between language environments that support blocks and such that do not as well as vice versa will therefore not be possible in the general case.

non-block => block

A migration from an environment that does not offer blocks to an environment that supports blocks can be performed in the context of the HLM migration mechanism. The objects of the source do not use that feature and can be transferred without problem. The interface declarations of the supported interfaces of the destination do not include the relevant detail of method code and can therefore be used by the negotiation algorithm.

block (>) non-block

A migration from an environment that supports blocks to an environment that does not will not be possible without changes to the destination environment. Since blocks can not be constructed from any other language constructs an native implementation built into the destination environment as well as appropriate changes to the destination language will be necessary.

block1 (>) block2

Whether migration will be possible between environments that implement blocks differently will depend on the individual case. Whether two implementations of the blocks concept can be matched or not can not be determined programmatically due to the halting problem. If a match can be determined manually the GOAL language has to be changes accordingly and the necessary adaptations can be implemented the context of the HLM migration mechanism.

exceptions

Many object based languages use concepts that generalize the management of error conditions within the sequence of statements. A construct called an *exception* in conjunction with an *exception-statement* like for example `try throw catch` can be used in languages like C++ [Str1991] or Java [GJS1996] to surround a number of statements that is susceptible to errors. Other environments like LOOPS [SB+1983] provide a similar construct named `Errorlevel`.

If an error occurs an *exception* is raised, the sequence of calls made so far is unwound up to the beginning of the exception-statement and the code handling of the exception, i.e. the *exception-handler* is executed as specified by the exception-statement. The flow of control usually continues after the exception-statement as the computations up to the occurrence of the error condition are aborted. For this reason exceptions are also called *non local return* control structures because throwing an exception does not return locally where it was thrown but execution continues after the exception-statement.

Exception statements can usually be nested and categorized by the kind of exception they handle. When exceptions are implemented as objects, they can be related by inheritance among their definitions. More general exceptions can be managed by default by outer exception-handlers while innermost exception expressions may be focused on particular problems like division-by-zero exceptions.

If an inheritance hierarchy of exceptions is available, it can be exploited to manage specialization among exceptions and exception handlers. Some exception mechanisms include provisions for an automatic retry of the operations within the exception expression, but in the general case it is up to the designer to surround the exception statement with a loop that retries the overall operation.

The GOAL language does not include an exception construct and the HLM migration mechanism does not support the use of exceptions as not all object based environments support exceptions. Non local return mechanism can not be recreated using other language constructs.

non-exceptions => exceptions

A migration from an environment that does not offer exceptions to an environment that implements exceptions can be performed in the context of the HLM migration mechanism as the objects to be migrated will not make use of the exception concept. The interface declaration used for the supported interfaces will not include the necessary detail.

However the compilation of the transferred interface definitions within the destination environment may require that appropriate default exception declarations are added to the native implementations of the objects to be transferred. The generation of the native object implementations within the destination environment have to be changes accordingly.

exceptions (>) non-exceptions

A migration from an environment that supports exceptions to an environment that does not support exceptions will not be possible at all. Since a exception-statement can not be recreated by any other language construct an native implementation of the exception concept is required for the destination environment.

exceptions1 (>) exceptions2

Migration between environments that support different exception mechanisms will not be possible in the general case. Whether the exception mechanisms can be matched can not be determined programmatically due to the halting problem. If a match can be determined manually the GOAL language has to be changed accordingly and the HLM migration mechanism need to be extended to perform the necessary adaptations.

pointers, garbage collection, finalization

Object based language environment can roughly be subdivided into such that use explicit memory references, also called *pointers,* like C++ [Str1991] and such that do not. Pointer based language environments require a designer to explicitly allocate and deallocate memory for objects while environments that use implicit references for all objects implement automatic memory management as for example Smalltalk [GoR1983] or Java [GJS1996].

The term *garbage collection* denotes the reclamation of resources that are no longer needed within environments with automatic memory management. Various algorithms like "mark and sweep", "reference counting", or "generation scavenging" and defragmentation have been developed to optimize the overall task.

Some object oriented environments augment the usual garbage collection mechanism through the concept of finalization. *Finalization* allows an object that is identified for reclamation to perform some housekeeping tasks before being collected. Alternatively the object can prevent its destruction within some environments.

Garbage collection is more complex in multithreaded environments where several threads of activities create, manipulate and destroy objects as well as in environments with active objects

that may not be subject to collection at all despite not being referenced. Even more complex is the distributed case where references to remote objects have to be considered (see page 179).

The GOAL language does not contain constructs for the explicit allocation or deallocation of objects and as a consequence the HLM migration mechanism assumes that automatic memory management is performed within the participating language environments. Language environments that offer pointers can only be used in the context of the HLM migration mechanism if features like pointer arithmetic are not used by the objects to be migrated.

garbage collection (=>) pointer

A migration from an environment with automatic memory management into an environment without such comfort will be impossible in the general case. Objects that would be collected at the source will not be deallocated explicitly at the destination and may outrun available resources.

The determination of the "deletion of the last reference" within the interface definitions transferred is impossible due to the halting problem. The only alternative - to ignore the necessary deallocation of migrated objects will only be possible in special cases as it will lead to the effect of memory leaks at least in the long term. However, this alternative may be sufficient in most practical situations.

pointer (=>) garbage collection

A migration from an environment with explicit memory management into an environment with garbage collection can be performed in the context of the HLM migration mechanism, provided that the objects to be migrated does not use pointer arithmetic. The native code generation has to substitute the explicit allocation and deallocation operations with object creation and null statement respectively.

Within the particular destination environments, modern garbage collection mechanisms should be able to cope with any dangling object structures that may result, especially cyclic ones. The use of pointer arithmetic on the other side can not be supported at all as matching operations of the destination environment can not be determined due to the problem of computability.

pointer1 (>) pointer2

Whether migration will be possible between environments that support different forms of pointer arithmetic depends on the individual case. A determination whether the pointer implementations of both environments can be matched will not be possible due to the problem of computability even in the case that both environments use pointers only in a type-safe fashion.

non-finalization => finalization

A migration from an environment that does not offer finalization to an environment that supports finalization can be performed in the context of the HLM migration mechanism. The objects to be migrated will not use finalization at all and the finalization methods of the objects of the destination environment do not have to be included in the interface declarations that are used to represent the supported interfaces.

Finalization (>) non-finalization

A migration form an environment that supports finalization to an environment that does not will not be possible in the general case. A determination of the “deletion of the last reference” that is necessary to insert the code of the finalization method into the non-finalized environment will not be possible due to the problem of computability.

finalization1 (>) finalization2

Whether a migration will be possible between environments that implement finalization in differently depends on the individual case. A determination whether two finalization mechanisms can be matched will not be possible programmatically due to the problem of computability. If a match can be found manually the GOAL language has to be changed accordingly and the HLM migration mechanism has to be extended to perform the necessary adaptations.

4.5 Object Migration among existing Language Environments

Migration between object based language environments can be performed in the context of the HLM migration mechanism if only a defined subset of the concepts of object based systems is used for the design of the objects to be migrated. Additional concepts can be supported using advanced migration techniques like adaptive migration or informed migration. Only few concepts will require changes to the destination environments prior to migration.

Migrateability is possible in most cases from an environment that does not support a certain concept to an environment that does. In such a case only a restricted subset of the functionality available at the destination can be used as supported interfaces for negotiation. For example parameterized types can not be used in negotiation but objects can be migrated from a non-parameterized environment to a parameterized environment.

Whether migration is possible form an environment that supports a certain concept to an environment that does not depends on the individual case. If restrictions are applied to the design of objects to be migrated the HLM migration mechanism will be able to perform migrations in most cases. In some cases simple enhancements of the HLM migration mechanism or the use adaptive or informed migration can enable migrateability. However, migration will not be possible in the general case if no restrictions to the design of objects are made. Table 4.b provides an overview of migrateability with regard to particular language concepts.

Concept			Concept
typed	(>)	(<)	untyped
hybrid	(=>)	(<=)	pure
parameterized types	(=>)	(<=)	parameterless types
class based	(=>)	(<=)	prototype based
single inheritance	(=>)	(<)	multiple inheritance
single inheritance	(=>)	(<)	delegation
multiple inheritance	(>)	(<)	delegation
shared	(>)	(<=)	not shared
protected	(=>)	<=	unprotected
instanciation	(=>)	(<=)	cloning
synchronous	(=>)	(<)	asynchronous
single threaded	(=>)	(<)	multi-threaded
reflective	(>)	(<=)	non reflective
blocks	(>)	(<=)	no blocks
exceptions	(>)	(<=)	no exceptions
garbage collection	(=>)	(<=)	no garbage collection
finalization	(>)	(<=)	no finalization

Table 4.b: The migrateability of objects with regard to particular language concepts (double arrows => indicate that an migration is possible using the HLM migration mechanism, parentheses around double arrows (=>) indicate that additional restrictions or enhancement are necessary, parentheses around an arrow-head (>) indicate that extensions to the HLM mechanism or to the participating environments are necessary.

Whether an existing language environment can participate in migration in the context of the HLM migration mechanism is determined by the prerequisites of migration, namely the use of type annotations, the ability to add interfaces at runtime, the ability to create objects from newly added interfaces at runtime and the support of communication means (see chapter 3 page 74).

The last precondition is met by almost all language environments either directly in the form of network transports or indirectly through the use of shared file system services. The first three preconditions are - surprisingly enough - only fulfilled by a small number of language environments.

Of twenty-six existing language environments that have been investigated in the context of the HLM migration mechanism only few fulfill all prerequisites and can participate in migrations. These language environments that can be called *migration-capable* differ in the set of concepts they support. Table 4.c provides a overview of the migration-capable languages and of the concepts they support and lists also a few prominent languages that can not participate.

	Actors (Act1)	Beta	**Clos**	C++	Eiffel	**Java**	Modula 3	**Objective C**	Omega	Python	Sather	Self	Smalltalk
type annotations	-	x	**(x)**	x	x	**x**	x	**x**	x	-	x	-	-
dynamic loading	(x)	-	**x**	(x)	-	**x**	-	**(x)**	-	x	-	x	x
dynamic object creation	x	-	**x**	-	-	**x**	-	**(x)**	(x)	x	x	x	(x)
hybrid/pure	p	(p)	**h**	h	(h)	**(h)**	h	**h**	(p)	h	h	(p)	p
parameterized	-	x	**-**	(x)	x	**-**	(x)	**-**	x	-	x	-	-
shared structure	-	x	**x**	x	x	**x**	x	**x**	x	x	(x)	x	x
shared state	-	(x)	**x**	x	(x)	**x**	-	**x**	x	x	x	x	x
protected state	-	(x)	**-**	x	x	**x**	-	**(x)**	x	-	x	x	(x)
class	-	x	**x**	x	x	**x**	(x)	**x**	-	x	x	(x)	x
instantiation	-	x	**x**	x	x	**x**	(x)	**x**	-	x	x	-	x
single inheritance	-	x	**x**	x	x	**x**	(x)	**x**	-	x	x	-	x
multiple inheritance	-	-	**x**	x	x	**-**	-	**-**	-	x	x	-	-
prototype	x	-	**-**	-	-	**-**	-	**-**	x	-	-	x	-
cloning	x	-	**-**	-	(x)	**-**	-	**-**	x	-	-	x	-
delegation	x	(x)	**-**	-	-	**(x)**	-	**(x)**	-	-	-	x	-
reflective	-	(x)	**x**	(x)	-	**(x)**	-	**-**	-	-	-	x	x
synchronous	-	x	**x**	x	x	**x**	x	**x**	x	x	x	x	x
multithreaded	(x)	(x)	**(x)**	(x)	-	**x**	x	**(x)**	-	x	(x)	x	x
asynchronous	x	-	**-**	-	-	**-**	-	**-**	-	-	-	-	-
exceptions	-	x	**x**	x	x	**x**	x	**-**	x	x	x	(x)	x
blocks	-	(x)	**(x)**	-	-	**-**	-	**-**	x	-	(x)	x	x
pointer	-	x	**-**	x	(x)	**-**	x	**-**	-	-	-	-	-
garbage collection	-	x	**x**	-	(x)	**x**	x	**x**	x	x	x	x	x
finalization	-	-	**-**	-	-	**x**	-	**-**	-	x	x	-	-

Table 4.c: An overview of existing object-based languages in the context of the HLM migration mechanism. Languages that are migration-capable are shown in a **bold** font (an x or a lowercase letter indicates that a language supports a concept while parentheses around an x or a lowercase letter indicates that a concept is not fully supported or only optionally supported or that it is available only within some specific language implementations).

Surprisingly, some well known languages are not able to participate in migration in the context of the HLM migration mechanism mainly because they do not support the indispensable prerequisites of the mechanism. The counter-examples that are listed in table 4.c are discussed in the following paragraphs in alphabetical order.

Actors The “Actor language” is not really a language but rather a design of an object-based system consisting of active objects that can be implemented in many languages including procedural ones. Actors have been included here as the only well known example of asynchronous message passing at the language level. Actors implement a prototype-based object model and allow the dynamic creation of objects. Unfortunately, Actors as defined by Agha [AgJ1999] do not use type annotations and can therefore not participate in migrations in the context of the HLM migration mechanism.

Beta The Beta language [MMN1993] is one of the most flexible object-based languages and offers the unique abstraction of patterns that effectively join objects, methods and even activation records into one seamless notion. Beta supports a form of delegation in addition to single inheritance as well as limited reflective and multi-threading capabilities. Unfortunately only a single implementation of the language exists and no mention of dynamic loading of object definitions or the dynamic creation of objects could be found in the literature.

C++ The C++ language [Str1998] is a very prominent if not the most well known object-based language, due to the fact that it is perceived as the object-oriented successor to the popular procedural language C. As a hybrid language C++ offers a number of features that are rooted in its procedural predecessor like pointer arithmetic and operator overloading. C++ also offers parameterized types in the form of templates and read-only reflective capabilities in the form of a runtime type system. A unique feature of C++ is the `friend` declaration for cross object access to state. The language can be extended to support multi-threading. Unfortunately, the C++ language does not offer dynamic creation of objects despite the fact that some language implementations can be extended to support dynamic loading of object definitions [IBM1993].

Eiffel The Eiffel language [Mey1992] is one of the most intensively studied object-based languages, a fact that results from the software engineering approach that motivated the design of the language. Eiffel offers a number of unique features, one being the declaration of assertions that allows for the automatic check of pre- and postconditions for the execution of methods. Unfortunately, no mechanism for the dynamic loading of object definitions or for the dynamic creation of objects could be found in the literature.

Modula 3 The Modula 3 language [BoW1995] is a hybrid language in the sense that it allows the definition of objects in addition to a procedural base-language. Modula 3 offers an extensive type-system and language primitives for the management of parallel threads. Unfortunately, no mechanism for the dynamic loading of object definitions or for the dynamic creation of objects could be found in the literature.

Omega The Omega language [Bla1994] is a not widely known prototype based language that offers strong typing and an inheritance like style of delegation. Omega offers parameterized types, an exceptions mechanism and blocks. Unfortunately only a single implementation exists that does not support dynamic loading of object definitions nor the dynamic creation of objects.

Python The Python language [LoF1997] is an interpreted object-based language that appears to be a good candidate for object migration that offers even thread-management. Python supports multiple inheritance, exceptions and even finalization. Unfortunately, Python does not support type annotations and can therefore not participate in migrations in the context of the HLM migration mechanism.

Sather The Sather language [StO1996] is a strongly typed object-based language that offers many unique features, for example block like control constructs called *iterators*. Sather supports parameterized

types as well as finalization and can be extended to support multi-threading. Despite the fact that Sather supports dynamic object creation, no mentioning of the dynamic loading of object definitions could be found in the literature.

Self The Self language [Sun1992] is a well known prototype-based language that offers multiple delegation relationships among prototypes. Self offers reflective capabilities, multithreading, a block concept that is also used for exceptions, and an abstractions called *traits* for sets of prototypes with common behavior that is used for optimizations. Self is able to dynamically load object definitions and to create objects dynamically but does not support type annotations.

Smalltalk The Smalltalk language [GoR1983] is probably the best known example of an object-oriented language. Smalltalk supports reflective capabilities, exceptions, blocks and multithreading. Smalltalk is able to dynamic load object definitions and can also support dynamic object creation through its reflective capabilities. Unfortunately, Smalltalk does not support type annotations.

More object-based languages have been investigated in the course of this work, for example Calico [BeL1993], Cecil [Cha1993], Dylan [SPM1992], Hybrid [Nie1987], Loops [SB+1983], Oberon 2 [Moe1993] and a number of others. These additional languages have not been included in this overview because not enough literature was available to decide whether these language fulfill the prerequisites of the HLM migration mechanism or not.

Whether the languages that are not able to participate in migration can be extended through changes of their definitions or their environments to support the prerequisites of the HLM migration mechanism has not been investigated. Although more languages may be able to participate in migration, changes to language definitions or environments lie clearly beyond the scope of this work as the HLM migration mechanism that is supposed to be working with existing languages rather then requiring language changes.

Migration-Capable Languages

The languages that support the prerequisites can all participate in the heterogeneous object migration based on the HLM migration mechanism. Two of these language environments, i.e. CLOS and Java have been used for the prototypical implementation of the HLM migration mechanism.

Since these languages support different sets of concepts migration of arbitrary objects is not possible among these environments. Only objects that have been restricted appropriately can be migrated in the context of the HLM migration mechanism. Support for additional language concepts can be added to the HLM migration mechanism but will be useful only for subsets of this collection of migration-capable languages.

The following paragraphs discuss each migration-capable language in more detail in terms of the concepts each language implements, how the HLM migration mechanism can be implemented within the particular language environment and how additional language concepts can be supported in the context of the individual language.

This overview is not intended as a list of facts about the migrateability of objects in the context of the individual language. This overview rather serves as a ranking of the probability of migration in the context of additional language concepts which have to be addressed individually by further research.

CLOS The Common Lisp Object System (CLOS) [Ste1990] is a well know object-oriented programming environment based on the imperative programming language Lisp[30] and is one of the few programming

[30] Modern Lisp dialects combine procedural techniques with the functional core of the language.

languages that has been standardized. CLOS is a hybrid language that integrates some but not all types of the underlying Lisp language into an object-oriented type system. CLOS is dynamically typed and offers optional type annotations for class and method definitions. Generic functions are used to implement methods in CLOS, which are executed in the context of the unique concept of method combination as defined by a totally ordered multiple inheritance lattice. CLOS supports exceptions as well as blocks and assumes a memory management that offers garbage collection. CLOS also implements a Meta-Object Protocol (MOP) that offers advanced reflective capabilities. CLOS offers interpretation as well as compilation of classes and methods and is able to dynamically load object definitions and to dynamically create objects. Some CLOS environments also implement multi-threading capabilities.

The prototypical implementation of the HLM migration mechanism within the CLOS language environment has to take the hybrid type-system in to account and generates the necessary conversion code. Type annotations have to be used by default and the generation of object definitions has to consider the limitations of generic functions, namely a consistent number of default arguments per generic function as well as the inability to use null values in message passing expression. The prototypical implementation of the HLM migration mechanism based on CLOS manages interface definitions as GOAL source files and uses the TCP/IP socket implementation of CLOS environments as a communication means.

Support for several additional language concepts offered by CLOS can be added to the HLM migration mechanism. The multiple inheritance lattice of CLOS has to be totally ordered, a condition that is checked by the CLOS compiler. CLOS therefore offers a good start for the extension of the HLM migration mechanism to support multiple inheritance. The strictness of CLOS in this regard can be used to value the multiple-inheritance lattice of other environments against it. The reflective capabilities of CLOS as defined by the Meta-Object Protocol [KRB1991] offers probably a superset of the reflective capabilities of other environments and can be used as basis for the experimental support for reflective capabilities by the HLM migration mechanism. The CLOS implementations of the concepts of exceptions as well as blocks are also flexible enough to define compatible subsets of their functionality that can be used for migrations with other environments.

A CLOS concept that can probably not be used in migration with any other environment is method combination. The definition of additional methods that are executed in a cascaded style before and after the main method that is defined by a message passing expression is unique to CLOS and to foreign for most other environments to be supported by an extension of the HLM migration mechanism.

The CLOS language flexible enough to be extended with additional language concepts. The concept of sharing and protection of state is partially supported by CLOS and depending on the implementation of garbage collection within the corresponding language environments, support for finalization can probably be

added too. CLOS is probably not a good candidate for support of the concepts of multi-threading or asynchronous message passing in the context of the HLM migration mechanism. Despite the fact that multi-threading is offered by some CLOS language environments the language does not offer any constructs to support it. Another concept that is not likely to be supported easily in CLOS are any sophisticated form of delegation, as the totally order multiple-inheritance lattice is the foundation of the CLOS language.

Java

The Java Language [GJS1996] is a well known object-oriented programming language that is widely used due to its foundation on a virtual-machine-definition that make Java programs hardware- and operating system independent. Java is a strong-typed class-based language that supports single inheritance and a separate inheritance concept of interfaces[31]. Java offers optimized primitive types that effectively make Java a hybrid language. Java supports exceptions and implements a built-in multi-threading model that is integrated with the definitions of classes and methods. Java is based on garbage collection and offers finalization. Java is able to load object definitions dynamically at runtime and to create objects dynamically at runtime. The Java language environment offers reflective capabilities via libraries.

The prototypical implementation of the HLM migration mechanism in Java has to take the differences between primitive types and Java objects into account an generates the necessary conversion code. The generation of object definitions has to consider that Java can not differentiate methods that can only be distinguished by their result type. The prototypical implementation manages interface definitions on the basis of GOAL source files and uses the TCP/IP objects offered by the Java libraries to establish communications.

Support for several additional language concepts offered by Java can be added to the HLM migration mechanism. The exception mechanism of Java makes use of the single inheritance relationships and can probably be used as a good start for the extensions of the HLM migration mechanism with support for exceptions. The same holds for multi-threading as the implementation of multi-threading within the Java language offers a good foundation for further experimentation in this regard up to asynchronous message passing in the context of distribution. Support for sharing and protection is also available within Java and can be used for migrations with other environment that support similar concepts.

Java language concept that can not be supported in the context of the HLM migration mechanism are the direct inheritance of interfaces that is unique to Java and can for example not be matched with the totally ordered multiple inheritance lattice of CLOS. Another concept that can not be supported unless it is available within the destination environment is finalization.

The Java language definition has been fixed as any extension to the language would require updates to the existing virtual machine implementations that are widely deployed. Java is therefore not a

[31] The `interface` keyword of Java should not be confused with the notion of interfaces in the context of the HLM migration mechanism.

good candidate for support of blocks, sophisticated delegation schemes or advanced reflective capabilities. Any concept that can not be supported on the basis of language libraries can not be used in migrations with the Java environment if extension are considered at all.

Objective C

The Objective C language is a known as a "former" candidate for the object-oriented successor to C. Objective C is an extension of the C language as it defines a hybrid, dynamically-typed class-based language that offers optional static typing for optimization purposes. Although a descendant of the C family Objective C offers more dynamic concepts then its sibling C++. Besides single inheritance Objective C also offers a delegation like concept of categories and protocols that allow additions to class definitions as well as methods which are independent of classes. Objective C supports protection of instance variables but not sharing of state via class variables. Objective C does not provide a exception mechanism but uses messages to signal errors among objects. A unique concept of Objective C is the possibility to define message passing expression dynamically, i.e. the selector as well as the hidden parameters of the message can be assigned at runtime, including assignments to the pseudo-variable `self`. This concept can be used to manipulate messages and to forward messages to different objects. Another unique concept of Objective C is the use of factory objects for the creation of objects via message rather then a special language constructs like new. Objective C supports the dynamic loading of object definitions and the dynamic creation of objects. Objective C enables but does not rely on the use of garbage collection mechanisms and does not support finalization. Objective C offers language support for asynchronous messaging but does not implement it. However, corresponding language environments support multi-threading.

An implementation of HLM migration mechanism in Objective C has to consider its hybrid nature that builds upon the type system of the C language. The generation of object definitions has to consider the keyword style of methods and message passing expression that is similar to Smalltalk. The creation of new objects via factory objects is also unusual and has to be accounted for in an implementation of the HLM migration mechanism. The message based error handling of Objective C has to be considered as well. Due to the absence of reflective capabilities, object definitions have to be handled as GOAL source files and a TCP/IP implementation is available within language environments as well.

Support for additional concepts of the Objective C language can be added to the HLM migration mechanism. Despite the fact that Objective C supports only single inheritance, the concept of dynamic message composition can be used as a starting point for the extension of the HLM migration mechanism to support delegation. The concept of protocols can be useful to support a subset of generic functions in the context of the HLM migration mechanism.

Language concepts that can not be supported in the context of an implementation of the HLM migration mechanism in Objective C are sharing and protection of state, exceptions and blocks, that are not well supported or not available at all respectively. The reflective capabilities of Objective C are also to limited to match

any other implementation of reflection. The concept of finalization is also not available within Objective C or within its language environments that support garbage collection.

The Objective C is based on the portable language C and can be extended comparatively easily. One likely candidate for an language extensions is the support of asynchronous messages that is already prepared for within the language through the `oneway` parameter qualifier for example. Especially Objective C language environments that support multi-threading are likely candidates in this regard.

This overview emphasizes the fact that has been proven by the prototypical implementation: the HLM migration mechanism is capable to migrate objects among existing object-based languages. Additionally a number of opportunities exist to support more language concepts in the context of the HLM migration mechanism.

Unfortunately, the majority of language concepts that are not already supported by the HLM migration mechanism require changes to the definitions of the objects to be supported. Only few language concepts can be supported with minor enhancements of the HLM migration mechanism and some concepts even require changes to the participating language environments which are effectively prohibitive.

Even if support for additional language concepts can be added to the HLM migration mechanism a partitioning of migrateability may result as not all participating environments are equally able to support additional language concepts even if the definitions of the objects to be migrated are changed through adaptive migration for example.

Furthermore, migrations that are performed in the context of additional language through the use of adaptations may no longer be symmetric or transitive. The adaptive changes can inhibit the inverse migration of an object if the adaptations applied are not recorded and can be inverted when necessary. Adaptation of interface definitions may also inhibit successive migrations to different destinations as well as subsequent migrations of other objects from the same source environment.

In hindsight, the status of migrateability is disillusionary at best. The lack of support for dynamism, i.e. applying changes to programs at runtime limits the use of object migration among existing language environments severely. The absence of type annotations is less severe as preprocessing tools can be used with most language environments to allow changes to source files that are filtered before native compilations but can be used for other purposes like the HLM migration mechanism.

The vast majority of object-based languages would be able to participate in migration if the basic prerequisites of the HLM migration mechanism could be met. Despite the fact that the diversity of languages limits migrateability if additional language concepts are supported, dynamic changes of programs in general and heterogeneous object migration in particular are treated negligently.

Besides the hint to language designer and implementers that dynamic features deserver more attention, the advantages of heterogeneous language migration and especially the benefits for the design of applications have to be emphasized more directly in order to change this detrimental situation persistently.

5 Applications of Heterogeneous Language Migration

A number of applications can be identified for heterogeneous language migration despite the fact that the HLM migration mechanism is certainly not sufficiently developed for general use at this point. The following overview of potential applications emphasize the possible benefits of heterogeneous language migration and are also intended to motivate further research.

Apart from technical issues that have been discussed in the previous chapter, a number of characteristics of the HLM migration mechanism influence its applicability and determine the benefits an application can achieve. The following sections offers a brief overview of the most influential characteristics of the HLM migration mechanism from an application standpoint.

- Semantics

  The HLM migration mechanism offers the biggest benefit where the semantics of a solution of part of it have to be conveyed to a different location or another environment. Especially applications that capture the knowledge of their users with programmatic means like macro or scripting languages will be able to transport that knowledge to another place or into a different context using the HLM migration mechanism.

- Heterogeneity

  Almost unnecessary to mention, the HLM migration mechanism will allow to exchange the status and the meaning of software artifacts among a broader range of applications as previously thought possible. The fact that all levels of heterogeneity can be addresses does not imply that all capabilities of the mechanism have to be applied for it to be useful. The HLM migration mechanism can be very beneficial among heterogeneous applications alone regardless of other levels of heterogeneity or homogeneity.

- Spontaneity

  The transfer of objects between software-systems with the use of the HLM migration mechanism requires no initial setup or special knowledge among the participating environments. Provided that an implementation of the HLM migration mechanism exists within each participating environment, any application that conforms to the design constraints of the migration architecture can transfer objects without further preparation.

- Independence

  A migration that is performed by the HLM migration mechanism leaves no dependencies between the participating environments. Furthermore, communication capabilities between the participating environments are only mandatory as long as the migration takes place. No permanent connection is required prior to or after a migration and the participating system may as well only communicate only for the course of a migration.

- Peer-to-Peer

  The whole architecture of the HLM migration mechanism is geared towards a cooperation of the participating environments rather then an exclusive control of either side. The migration algorithm is performed by the source environment but only with the cooperation of the destination environment. If full implementations of the HLM migration mechanisms exists on either side, migrations can be performed symmetrically any time. As a consequence the HLM migration mechanism can be characterized as a peer-to-peer system, as all participating environment have equal rights. Especially no central server that coordinates the activities is necessary.

- Responsiveness

  The nature of the HLM migration mechanism allows for comparatively short-term reactions to a need to convey the status and meaning of a software artifact to another environment. Although the prototypical implementation of the HLM migration mechanism is not particularly optimized for performance, the overall task of transferring an object to another environment is performed quickly if compared to the otherwise necessary manual activity.

In summary, the HLM migration mechanism provides the best benefit in situations where the semantics of software artifacts has to be exchanged between heterogeneous systems in a short timeframe that does not allow for extensive preparations neither of technical nor organizational nature. The characteristic of independence does not only allow for ad hoc operations of the HLM migration mechanism but also enables the support of mobile and wireless devices which can be used disconnectedly.

In an opposite position, a number of factors are of less importance for the applicability of the HLM migration mechanism. The distance between the participating environments as well as the available bandwidth do only affect the use of the HLM migration mechanism neither does the synchronous or asynchronous nature of the communication medium.

Although the overall performance will be increased if a higher bandwidth can be utilized, the applicability of the HLM migration mechanism does not depend on the available bandwidth. Migrations can be performed using high speed networks as well as with low bandwidth connections over for example wireless networks.

Even synchronous connectivity is not strictly necessary for the operation of the HLM migration mechanism. An asynchronous communication medium will be sufficient as long as the delivery of messages between the participating environments can be guaranteed. For example message queuing services can be used by the HLM migration mechanism.

The HLM migration mechanism is also largely independent of the application domain as well as of the complexity of the objects that have to be transferred. The purpose the HLM migration mechanism is used for, e.g. financial transactions, industrial engineering, educational knowledge-sharing does not affect the operations of the HLM migration mechanism.

The complexity of the objects to be migrated does also not directly affect the applicability of the HLM migration mechanism. The number of objects and the complexity of their functionality as well as of their relationships has only an impact on the overall time it takes to process a migration but does not influence the question whether the HLM migration mechanism can be used for a particular task or not.

Limiting Factors

The applicability of the HLM migration mechanism is limited by the scope of the extensions that can be applied to the destination environments as well as indirectly by the overall timeframe within which the necessary changes have to be available. Both of these dimensions define upper as well as lower bounds for the applicability of the HLM migration mechanism.

The migration of atomic objects among heterogeneous environments at the language level is the design goal of the HLM migration mechanism. The mechanism is therefore limited to the transfer of subsets of the collection of objects that define an application, or of partial functionality of underlying libraries.

On the other hand an implementation and use of the HLM migration mechanism makes only sense when the task of migrating objects appears comparatively often and has to be performed in a relatively short timeframe. If only predefined functionality has to be transferred and the available time is not a primary factor, a conventional approach like a porting effort of the necessary functionality will be more economic. The HLM migration mechanism is not a tool that aids in the transfer of whole applications or operating systems.

Apart from being able to transfer semantics among heterogeneous environments the HLM migration mechanism is most beneficial in situations that require a comparatively short-term reaction to a need to transfer semantics among environments. Figure 5.a shows the correlation between the scope and available time of the changes necessary to perform a migration.

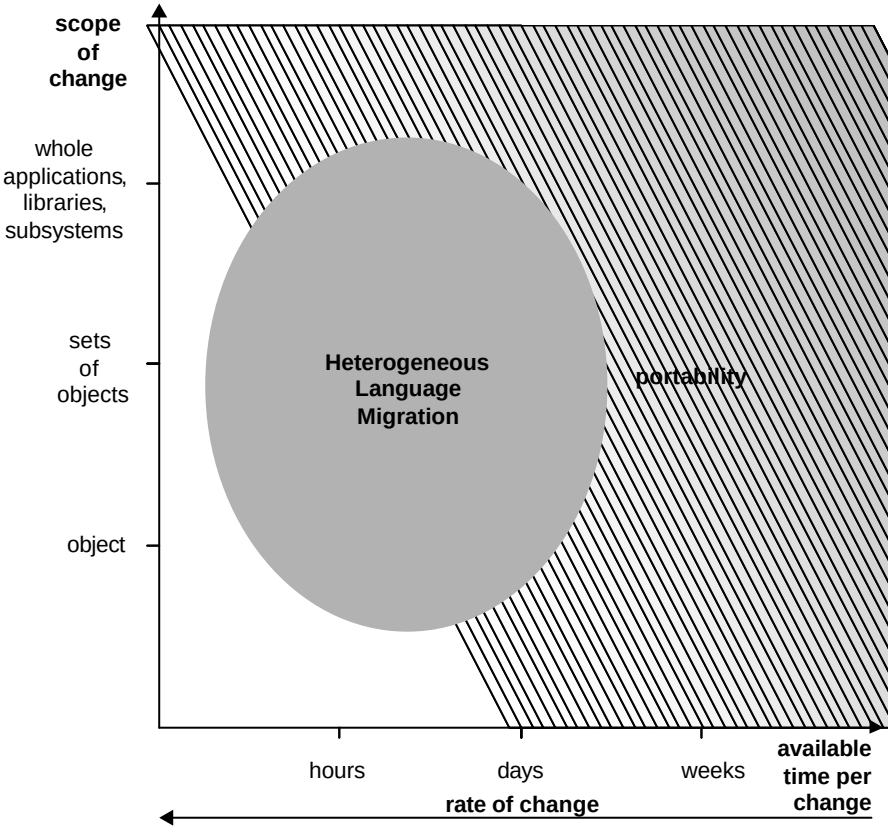

Figure 5.a: The use of the HLM migration mechanism makes sense when the rate of change that is reflected is the frequency of migrations is high and the scope of the changes can be handled by the mechanism. If more time is available or more fundamental functionality needs to be transferred a conventional approach like a porting and integration effort will probably be more economical.

On the other hand, the HLM migration mechanism is not and will probably never be able to perform high volumes of migrations in a very short time. Sub-second migrations of huge numbers of objects are very unlikely, even if the same interface set is involved. Likewise the transfer of subparts of objects like the functionality of a single method lies out of the scope of applicability of the HLM migration mechanism (see also chapter 4 page 178).

Heterogeneous object migration is in some sense competing with traditional software development techniques that can be applied if short term reaction and spontaneity of transfer do not have to be emphasized. However, the HLM migration mechanism can also be used in the context of software engineering, e.g. within software design tools to exchange independently developed software models (see also page 226).

The following example try to illustrate the scope of applicability of the HLM migration mechanism in practice. The first example describes a scenario that combines several characteristics of the HLM migration mechanism to emphasizes the possible benefits for a new kind of application.

The subsequent examples describe other interesting areas of usage more briefly. The use of the HLM migration mechanism to convey dynamic content is discussed in the context of hypermedia authoring. A match of different software designs is described in an example of software engineering. The debugging of distributed system is used to illustrate the application of migration for software development. The technology of mobile agents can benefit from heterogeneous migration and administrative tasks like network management can take advantage as well.

5.1 Distributed Collaboration

Probably the biggest benefit of the HLM migration mechanism from an applications standpoint is the fact that it enables a new advanced form of collaboration. The transfer of semantics of objects opens the way to not only cut and paste static data like text and graphics but to capture, exchange, combine and evolve the dynamics of ideas and knowledge that could not be transported before.

Previously, the inherent dynamics of real world phenomena could only be conveyed as static tables of measurements, as presentations graphics, reports, or books. Even if the changing nature of such phenomena can be captured in software models these models can not be communicated freely as they are restrained to the computer platforms they were developed for.

Moreover, the thinking and creative work of humans has been oriented towards static tangible artifacts since the invention of typography or even script due to the fact that the dynamics of human ideas could not be conveyed through any medium. This situation is changing slowly since computers and the Internet are in wide-spread use.

Today the dynamics of human ideas can be captured to some extent through software development, but their communication among humans is still limited through the new barriers of heterogeneity. The HLM migration mechanism provides a first step to alter that and to open up new ways to exchange and combine ideas and knowledge.

Ideas and phenomena that have been captured as software objects can be exchanged freely using the HLM migration mechanism. In combination with object-based end-user scripting languages, independently developed objects can be integrated to higher order objects as the ideas they represent are combined and can be evolved to new knowledge.

The HLM migration mechanism in its present state has a long way to go to be able to support such an advances usage. Three scenarios can illustrate the usage of the HLM migration mechanism in this context before the necessary enhancements to the mechanism as well as other details of its implementation are discussed.

Research Scenario

If for example a evolutionist develops a software model that describes the extinction of species in a statistical tool, he can migrate that model using the HLM migration mechanism to a the simulation tool of a biologist, who will be able to test it against current data from a particular endangered species. The biologist will also be able to modify and combine the model with some of his own work through scripting and to migrate the combined model to a mathematician who can validate it for soundness.

Engineering Scenario

In the light of ever shortening product development cycles, the changes to the construction of a car in a graphical design tool can be reflected more timely in the planning tool for the corresponding manufacturing plant if the designer of the automobile and the factory designer can exchange the model of the underlying construction seamlessly. Extensions to the model made on either side can be migrated using the HLM migration mechanism to the corresponding co-workers.

Business Scenario

A complex deal for the financing of a large construction project can be put together more easily if the different stakeholders can coordinate their divergent interests in a multidimensional “what if” model of the whole deal. Each participating financial institution can migrate its constraints as

an object to the leading institute or a clearinghouse that reconciles the various conflicts. Each stakeholder will be able to run his own analysis against the combined model if that is migrated back as a copy. Refinements of the individual contributions can be added and disseminated in the same way.

Technical Background

The sharing and reuse of ideas captured as objects should be a simple and seamless as possible for the end-user. Based on the graphical user interfaces of today a user should be able to open a window of an application that is running on a remote machine to drag and drop objects that are found there into his own local applications.

The corresponding migrations including the necessary negotiation will be performed by the HLM migration mechanism in the background. After the migration the user should be able to manipulate and work with the migrated objects as if they were constructed natively within the local application in the first place.

In order to integrate locally created and migrated objects a visual programming tool could be used in the ideal case to indicate simple message passing as well as the use of parameters among objects. The construction of more complex objects or integrations can be done with end-user scripting tools that use all elements of the corresponding interfaces.

Distributed Usage

Due to the spontaneous nature of the HLM migration mechanism the collaboration of individuals as described above can be initiated across organizational boundaries without preliminary setups or configurations. The HLM migration mechanism enables a free flow of ideas that are captured as objects as well as the combination of their dynamic behavior.

This exchange can be performed through any medium, via local area networks as well as via the Internet and even through wireless communication services. Due to the support for heterogeneity by the HLM migration mechanism potentially any computer hardware and object-based software can be included in the exchange of objects.

Mobility Support

Since the HML migration mechanism creates no dependencies among the participating environments mobile devices can be supported as well. Notebooks, Personal Digital Assistants (PDA) and eventually even advanced mobile phones will be able to participate in the migration of objects and can operate disconnectedly on their own after objects have been migrated.

In the context of collaboration, knowledge workers will be able to migrate objects onto their notebooks, change these objects at customer sites or during plane flights and migrate the objects to the workstations of coworkers. The HLM migration mechanism explicitly supports the disconnected use of the migrated objects, provided that these objects do not require themselves, for example a connection to the Internet.

Collaboration can also be initiated ad hoc among virtual teams of knowledge workers that are assembled so solve specific problems within extremely tight schedules. The HLM migration mechanism does support collaboration via the Internet without the use of a central server and regardless of the form or location of the Internet connection used.

Collaborative Applications

In order to support the migration and combination of objects that represent ideas, collaborative applications have to be based on a language environment that can be employed by users to capture ideas as objects and that implements the HLM migration mechanism. The language environment will also expose the semantics of migrated objects and can be used to combine migrated objects with local ones to form new higher order objects.

The HLM migration mechanism has to be enhanced in several ways in order to support the collaborative applications described above. Standard interfaces for graphical user interfaces have to be defined including support for drag and drop. The management of interface definitions has to be integrated with the language environment particular collaborative application as well.

Additional enhancements of the HLM migration mechanism can also simplify the usage of collaborative applications. For example a versioning mechanism for interface definitions will ease the evolutionary use of migrated objects as independent changes to existing interfaces can be managed.

The security of the collaborative applications can be addressed through a check of migrated objects for malicious functionality. The opposite check, whether a destination environment may act maliciously against a migrated object is not relevant as the object is migrated completely into the control of the destination environment anyway.

Probably the hardest problem with regard to collaborative applications will be support for secrecy by the HLM migration mechanism in order to solve for example copyright issues. As a migrated object may include intellectual property in the form of algorithms or data, neither source code representations nor a translation into the language of the destination environments are possible. The problems of security and secrecy provide opportunities for further research.

Despite the pervasiveness of today's applications that are oriented to static data some innovative work in the direction described above has already been done in the Openstep [Cra1996]; Penpoint [CaS1991], SOM [IBM1993] and Taligent [CoP1995] environments, which unfortunately did not prevail in the commercial marketplace.

The use of migration for mobile devices has been proposed for the homogeneous case in the agent systems LIME [PMR1999], MASE [KRR1998], TACOMA Lite [Joh1998] and TNET [HCS1997]. Even the use of adaptation has been proposed to address the differences between stationary and mobile systems [Sat1998].

5.2 Hypermedia Authoring

The term "hypertext" coined by Ted Nelson [Nel1987] denotes the interconnection of textual information with references that can be used to jump from the representation of one piece of text to a referenced piece of text. The best know example for such a system is the World Wide Web that presents pages formatted in the Hypertext Markup Language and allows the user to access a referenced page upon the click of a button via the Internet.

Hypermedia systems extend the notion of hypertext to all kinds of media as well as to software artifacts. A hypermedia reference may not only point to a piece of text but for example to a sequence of frames of a motion picture or even to an event in a simulation. Simple multimedia extensions like graphics, animations, sound and video can be embedded in web-pages as well. But full hypermedia capabilities that are seamlessly integrated are only available within experimental and specifically designed systems like Intermedia [YS+1988] that are often implemented using object-technology.

Java applets [Sun1999] can be used to extend the hypertext metaphor. The applet technology transfers the necessary functionality and some parameters to the requesting web-browser but is only able to create new objects locally within the browser. The content presented by applets in general is in most cases read-only although network connections can be established by applets within defined limitations.

An integration of the HLM migration mechanism into web-browsers will allow existing objects to be migrated to web-browsers including their state and behavior. The migrated objects are fully functional and can also include an edit mode that lets the user change their contents or appearance. The modified objects can then be migrated back to their origin in order to make those changes persistent.

Alternatively, the changed objects can be migrated to a different server-systems in order to implement for example a review workflow among a team of editors. Which alternative is chosen does only depend upon the initiation of the appropriate migration requests from within the web-browser.

Consequently, an interactive hypermedia authoring environment can be build on top of existing technology that can be extended to new media types with comparatively low effort. The HLM migration mechanism will have to be implemented within the participating server environments and web-browsers using existing scripting languages or embedded Java environments.

The multimedia objects to be used will have to be designed to be migrateable and appropriate standard interfaces have to be defined to access the display and interaction functionality provided by web-browsers. The user interfaces of the multimedia objects will allow read-only as well as editing access. The web-browsers will have to implement a menu command that can be used to migrate the changed objects back to the server.

The hypermedia extension described represent the minimal extensions that have to be made to implement a web-browser based hypermedia system. Although a hypermedia system based solely on the HLM migration mechanism may be able to provide more functionality an integration based on existing Internet standards seems more appropriate.

5.3 Mobile Agents

Software entities that decide for themselves where to move to between nodes of a network are called *mobile agents* or *autonomous agents*[32] [Wei1999]. Mobile agents are created within a source system for a specific purpose and released into a network of nodes that are able to host them.

Mobile agents do not necessarily have to return to their point of origin but may perform their work continuously or exit silently when finished. Some systems are able to let agents cooperate in fulfilling their work because complex tasks can be performed through a suitable combination of comparatively simple agents [MB+1998].

Mobile agent technology is tightly related to object migration which can be used as a base technology for the implementation of mobile agents [Kna1996]. The design of agents systems has also let to the development of new migration techniques [BGP1997]. However, most existing agent systems are implemented in the context of homogeneous software environments.

Many agent systems are based on virtual machine technology, especially the Java environment [GJS1996] as for example Voyager [Gla1998]. Only few agents system address heterogeneity independently of virtual machines , for example through the requirement of new language implementations instead like Ara [PeiS997].

Apart from being able to overcome the differences of the hardware, operating systems and languages used the HLM migration mechanism can be especially beneficial for the implementation of mobile agents as the migration mechanism works with existing language environments.

The functionality of the agent system to be implemented for each participating platform in addition to the HLM migration mechanism can be minimized, as only the necessary functionality for an agent to execute will be added to a target environment. The developer of an agent system can concentrate on the agent functionality and has to consider the mobility aspect only as far as the migration architecture of the HLM migration mechanism is concerned.

In order to support agent migration, the HLM migration mechanism has to be implemented for the participating environments. Additionally some specific standard interfaces will have to be defined for the interaction with the agent system that needs to be implemented by the participating environments as well.

The agents themselves can be implemented as single objects or small groups of dependent objects. Unfortunately, agents usually self-initiate their migration using a `go()` statement or some similar mechanism. Such an initiation can not be used in the context of the HLM migration mechanism, as the objects to be migrated have to be inactive.

An agent implemented on top of the HLM migration mechanism may however register a request to be migrated with a modified `OM_Porter` object which queues the request and services them one at a time. This restriction does not appear to be to limiting as agent systems tend to provide hundreds of agents and individual migration request will probably have to be queued anyway.

The general pros and cons of agents are discussed by Chess et al. [CHK1996] and migration between heterogeneous agent environments has also been discussed [GSC2000]. The transfer

[32] The term autonomous agent is also used for robot systems that are able to perform task without help.

of functionality has already been proposed [Pal1997] in the context of homogeneous systems including repository based solutions [AcS1996] to the problem.

5.4 Team Development

An almost nature application for the HLM migration mechanism seems to be the support of software development teams. As object-based software project are often complex and implemented by a team of software engineers, each engineer will have to develop some interfaces on his own and integrate his contribution into the overall solution later.

The HLM migration mechanism can be used to test the independently developed interfaces whether they can be integrated or not. However, this would only make sense for applications that are already developed for the HLM migration mechanism as the overhead to provide migrateability for the object to be tested would be prohibitive.

The HLM migration mechanism can nevertheless be beneficial in a different area of software development. An analysis and design tool can be implemented using the HLM migration mechanism in such a way that independently developed parts of a software design can be combined through migration of the objects that represent the corresponding formal specifications.

If for example a design is based on diagrams of the Universal Modeling Language (UML) the objects that represent the various symbols of the UML diagram can be migrated between design tools of different vendors. The independently developed software designs can then be combined into a single UML diagram.

As UML diagrams are standardized, a standard interfaces of the HLM migration mechanism will have to be defined for each element of an UML diagram. These standard interfaces can then be implemented through wrappers within the different design tools that also have to implement the HLM migration mechanism natively.

A migration of an UML diagram between design tools of different vendors will then only transfer the state of the objects that define the diagram within the source environment as their dependent interfaces will all be already supported by the destination, i.e. the design tool of another vendor natively.

Alternatively a formalized textual representation of the elements of the UML diagrams can be used in place of the GOAL language and the negotiation algorithm can be used to match differences of UML diagrams directly. Unfortunately, such an approach would require an complete reimplementation of the migration mechanism, that would thus also be confined to the migration of UML diagrams.

5.5 Distributed Debugging

Software systems that are spread over a number of network nodes are difficult to debug. The nature of networks, especially the communication latency as well as the independence of the processing nodes imply a great range of possible problems. Object technology can simplify the design of distributed systems but does not eliminate their inherent problems.

Difficult problems like sub-optimal performance can persist for semantically correct distributed software systems even if network management is enforced. Due to subtle race conditions among network components, performance tuning of distributed applications conducted while running the distributed program in question mighty not be successful, depending in the circumstances.

Debugging applications over the network changes the pattern of network conditions these applications experience and may thus prevent the identification of certain problems for example network performance bottlenecks. While such a problem can not be diagnosed with debugging enabled it may still exist when the application is run without debugging.

One solution to the problem of distributed debugging could be the use of debugging probes that can migrate from node to node isolating remaining problems that other debugging facilities have failed to identify. The debugging probes will be able to gather debugging information without

requiring network communication and with only minimal impact on local systems during debugging.

The gathered information can be transferred to a central debugging console after a test phase has been conducted or direct action can be taken locally at the debugging site without the need to contact central administrators. Depending on the nature of the application to be debugged and of the debugging tools, the debugging probe may itself be able to migrate closer to the potential source of a problem.

Besides overcoming the usual differences of hardware and operating systems that are the norm in distributed systems today, the use of the HLM migration mechanism to implement debugging probes will probably be able to significantly decrease the debugging time in situations like to one describe above.

Being able to move debugging functionality to the location of potential problems will help to shorten the turn round cycle that is necessary for the effective debugging of the complex behavior of distributed systems. This is especially true for heterogeneous systems as the debugging functionality does not have to be ported manually to all participating environments.

In order to use HM migration mechanism for distributed debugging purposes the concept of a callback mechanism needs to be supported through appropriate standard interfaces. The program to be debugged needs to be augmented with code instrumentation to send debugging information in the form of callback messages to a migrateable debugger object.

The HLM migration mechanism will have to be implemented within the participating distributed environments. Depending on the specific problem encountered a universal debugging probe can be migrated to the suspicious systems that will just gather a log of events for later analysis. More specific functionality like a control of network parameters or the emission of a signal if certain thresholds are surpassed can be implemented individually.

The use of HLM migration mechanism for distributed debugging will enable a more interactive style of debugging. If the debugging probes are implemented as a modular set of dependent objects modifications of the debugging functionality can be migrate e.g. in the form of additional analysis objects more frequently and will allow for a quicker isolation of problems within an distributed system.

The presented notion of an autonomous mobile distributed debugger that represents a crossover of object migration and mobile agent technology has already been prototyped for the tcl based agent system AGNI [Ra+1998] in the context of homogeneous migration among procedural language environments.

5.6 Network-Management

In the age of the Internet computer networks are constructed from a multitude of devices supplied by different manufacturers. Stationary network routers, telecommunication equipment, satellites, wireless relay stations, and all kinds of mobile devices are some components of the increasingly complex spectrum of networking technology.

The heterogeneous nature of the components that build today's network makes it difficult to maintain the quality of service that is expected. At present networks are managed through the use of network management protocols that send a periodic flow of status information through a hierarchically structured lattice of managed devices to a central management console. Commands and configuration information is sent downstream in order to take action when problems arise.

Using this common technique a network manager relies on the network itself to communicate management information. This widely used approach can be problematic in critical situations even if out of band communication channels are used. While inevitable if no separate communication links are available the added communication decreases the available network bandwidth and the probability that management commands get through decreases further if the network component to be managed is already near congestion.

Despite the fact that the design of today's network management tools incorporate object technology to represent network devices, the technology of objects is not used to its full potential. With the help of the HLM migration mechanism a different approach to network management can be implemented.

Rather then gathering network information centrally and sending commands that in critical situations may never arrive to remote network devices the devices themselves can be enabled to act. Management directives can be implemented as objects that are migrated to distributed network devices. The objects can manage these devices independently even in the presence of network failures.

The management objects to be migrated and their encapsulated decision criteria can be implemented centrally and distributed among diverse devices using the HLM migration mechanism. Differences of hardware and operating-systems of the various devices involved can be overcome in the context of the HLM migration mechanism through a cross-compilation of the interface definition to suitable native code that is transferred directly to the devices during the transfer of semantics phase.

The migration of objects with new directives from a central management systems to the distributed heterogeneous devices can be accomplished using cross-compilation of each of the different hardware and software platforms involved. The available functionality of a particular device will need to be determined using a modified version of the negotiation algorithm that also works with compiled executables.

The migrated management objects will be able to manage their respective devices even if a central management facility can not be reached. Logically centralized changes of the directives can be distributed quickly among the diverse set of participating devices through the management of sets of devices with equal functionality. A cross-compilation will only be necessary once for each set not for each device and distributed compile servers can be used to provide the necessary language support.

In a related approach the HLM migration mechanism can be used to distribute active network probes for the purpose of preventive maintenance of network devices. Objects that implement testing and upgrading routines can move from node to node within an network, gather quality of service statistics, report potential problems and perform maintenance tasks like software upgrades.

Using the HLM migration mechanism in combination with cross-compilation network probes will be able to move between different network devices and perform configuration and maintenance work even in situations where for example management communication from the outside is no longer possible due to a device misconfiguration. A network probe object that resides at such a device while the misconfiguration happens will be able to determine the communication breakdown and reset the configuration of the device to the last known functional configuration.

The use of both kinds of migratable network management objects will probably lower the number of cases where an on-site action of human personnel will be necessary significantly. Through preventive maintenance the need to actually visit a site for maintenance work can probably be limited to pure hardware related problems.

The use of the HLM migration mechanism will allow the enforcement of a single standard for the management of the quality of service and the maintenance procedures across diverse devices. The negotiation mechanism can also be used to keep object based network operating and management software up to date among heterogeneous network equipment.

The use of migration technology for network management has be analyzed quantitatively in the in the context of mobile agents [BaP1998] identifying several areas of potential benefits. In the context of homogeneous migration the use of object or agent migration has already been proposed for high speed ATM networks [HaB1998], for mobile network management [SaM1999], for UTMS networks [KüP1998] as well as for the deployment of network services in general [KrS1999].

6 Summary and Perspective

The work at hand addresses heterogeneous object migration among existing language environments. It provides a detailed analysis of existing migration systems that determines similarities as well as differences among the various implementations. A characterization of the possible levels of heterogeneity prepares the basis for a discussion of the existing as well as the potential approaches to overcome heterogeneity.

The analysis of existing migration systems reveals that migration between heterogeneous environments is only insufficiently supported. Existing approaches to heterogeneous migration at the language level require significant changes to existing language environments or require new language implementations altogether.

As the main contribution of this work a novel migration mechanism is designed that is able to migrate objects including their semantics between heterogeneous language environments without changes to the definitions of the migrated objects during migration or to the participating environments prior to migration. A prototypical implementation of the mechanism is developed and a sample application provides a proof of concept.

The design of the mechanism called Heterogeneous Language Migration or HLM consists of a migration architecture that enables the determination of sets of dependent objects that have to be migrated collectively. The extraction of objects out of running applications is achieved through an appropriate design of the application objects while avoiding the need for support from language runtime environments.

The HLM migration mechanism uses a negotiation algorithm that determines which part of the semantics the objects to be migrated depend upon is already supported by a particular destination environment and which part needs to be transferred to the destination environment. Source code abstractions are used for the transfer of the semantics and the state of objects.

Based on a number of common language concepts the HLM migration mechanism is able to migrate objects that adhere to the migration architecture between existing heterogeneous language environments. The HLM migration mechanism is implementation prototypically for the Java and CLOS languages and it can also be ported to other environments effortlessly.

In order to aid the developer in the design of applications of the HLM migration mechanism some guidelines for the construction of applications are given and development tools that check the preconditions of migration are proposed. A sample application is used to demonstrate the feasibility of the HLM migration mechanism as well as its benefits in practical situations.

As the HLM migration mechanism is limited by some implementation constraints as well as by the objectives various augmentations of the mechanism can be proposed. An analysis of the characteristics of the HLM migration mechanism is provided as a basis for the discussion of possible augmentations. Enhancements that follow the design of the mechanism as well as extensions that overcome the constraints implied by the objectives are suggested.

Enhancements of the HLM migration mechanism are able to optimize consecutive migrations, add further standard interfaces, enlarge the applicability of the mechanism to objects that can be encapsulated and add support for location independence through the use of proxy objects. These enhancements illustrate the scope of the HLM migration mechanism. It is applicable in most practical cases although the design of applications has to be changed accordingly.

Extensions to the HLM migration mechanism are proposed to challenge the limitations implied by the objectives. Adaptations of objects to particular destination environments and language concepts can be implemented if objects are allowed to be changed during migration. New forms of the negotiation and the migration algorithm used are discussed in the context of this technique.

As changes to the participating language environment are considered, an extension to the HLM migration mechanism that supports the transfer of computations is described in detail. Additional language concepts can be supported through enhancements and extensions of the HLM migration as well although some will require prohibitively complex implementations. An analysis of existing languages reveals that only few comply with the prerequisites of migration in general while most could participate in migrations based on the HLM migration mechanism in principle.

The scope of applicability of the HLM migration mechanism as well as potential benefits of its usage are discussed. A new kind of collaborative applications can be created through the use of the HLM migration mechanism and a number of other promising application areas have been identified as well.

Future Research

Due to the limitations of the HLM migration mechanism two main areas of further research can be identified. The prototypical implementation of the mechanism can be improved significantly as such changes will be necessary to gain broader experience with applications of heterogeneous language migration.

Although a number of enhancements can be implemented in the context of the prototypical implementation, some fundamental extensions to the HLM migration mechanism will nevertheless require a reimplementation of the mechanism. A split into several code streams for the investigation of different extensions appears to be inevitable.

In order to make the prototypical implementation of the HLM migration mechanism more practical and to achieve better performance, the language processing capabilities have to be improved and the support for additional standard interfaces, especially support for graphical user interface has to be added.

Experiments with variations of the negotiation algorithm as well as with adaptive migration can be performed on the basis of the existing implementation. This includes structural interface equivalence and Informed migration. The prototypical implementation will have to be enhanced with the necessary analytical and adaptive capabilities though.

A more challenging task will be the implementation of support for proxy objects that can be combined with fragmented migration and eventually support for a combination of migration and replication. These extensions of the HLM migration mechanism can still be investigated in the context of the prototypical implementation although additional language processing capabilities will be required to generate sovereign and proxy interfaces.

Whether the support for heterogeneous computations in the context of the HLM migration mechanism will be investigated remains questionable as significant changes to participating language environments will be necessary. Such changes may be possible in the context of interpretive language environments though.

Apart from these obvious research challenges a number of side-problems can be investigated as well. A versioning mechanism for interfaces can be combined with the HLM migration mechanism and will be beneficial for several application areas. Security and secrecy issues will also have to be investigated in order to be able to use an advanced version of the HLM migration mechanism in practice.

Appendix

The following additional information is available:

A Syntax-Notation

The following notation is used to specify the syntax:

(a)	grouping of syntactical rules
a b c ...	a sequence of syntactical elements
a | b | c ...	one of the given alternatives may apply
{ a }	an optional element
{ a },*	the element may occur zero or more times separated by the given separator, which in the example here is a comma , sign
{ a };$^{+}$	the element may occur one or more times separated by the given separator, which in the example here is a semicolon ; sign
non-terminal	non-terminals are shown in an *italics font*
`terminal`	terminal are shown in a **`bold fixed width font`**

B GOAL Syntax

The Generalized Object Abstraction Language (ORL) has the following syntax:

interface_declaration	**`interface`** *interface_identifier* { **`:`** *interface_identifier* } (**`{`** { *component* }**`;`*** { *signature_declaration* }**`;`*** **`};`** | **`;`**)
interface_definition	**`interface`** *interface_identifier* { **`:`** *interface_identifier* } (**`{`** { *component* }**`;`*** { *signature* }**`;`*** **`};`** | **`;`**)
component	*component_interface component_identifier*
component_interface	*interface_identifier*
signature_declaration	*signature_interface signature_identifier* **`(`** { *parameter* }**`,`*** **`)`** { **`{`** { *variable* }**`;`**$^{+}$ **`}`** } **`;`**
signature	*signature_interface signature_identifier* **`(`** { *parameter* }**`,`*** **`)`** **`{`** { *variable* }**`;`*** { *statement* }**`;`**$^{+}$ **`} ;`**
signature_interface	*interface_identifier* | **`void`**
parameter	*parameter_interface parameter_identifier*
parameter_interface	*interface_identifier*
variable	*variable_interface variable_identifier*
variable_interface	*interface_identifier*
statement	*if* | *while* | *do* | *return* | *assignment* | *message_send* | **`{`** { *statement* }**`;`*** **`}`**
if	**`if (`** *expression* **`) then`** *statement* { **`else`** *statement* }
while	**`while (`** *expression* **`)`** *statement*
do	**`do`** *statement* **`while (`** *expression* **`)`**
return	**`return`** *expression*
assignment	*assignee* **`=`** *expression*
assignee	*variable_identifier*
message_send	*recipient* { *message* } *
recipient	**`this`** | *parameter_identifier* | *variable_identifier*
message	**`.`** *signature_identifier* **`(`** { *expression* }**`,`*** **`)`**
expression	**`-`** *sub_expression* | **`!`** *sub_expression* | *sub_expression* **`+`** *sub_expression* | *sub_expression* **`-`** *sub_expression* | *sub_expression* **`*`** *sub_expression* | *sub_expression* **`/`** *sub_expression* | *sub_expression* **`&`** *sub_expression* |

	sub_expression **|** *sub_expression* | *sub_expression* **<** *sub_expression* | *sub_expression* **>** *sub_expression* | *sub_expression* **<=** *sub_expression* | *sub_expression* **=>** *sub_expression* | *sub_expression* **==** *sub_expression* | *sub_expression* **!=** *sub_expression* | *sub_expression*
sub_expression	*constant* | *instanciation* | { **(** *interface_identifier* **)** } *access* | { **(** *interface_identifier* **)** } *message_send* | **(** *expression* **)**
access	**this** | **super** | *variable_identifier* | *parameter_identifier*
instanciation	**new** *interface_identifier* **( )**
constant	*boolean_constant* | *integer_constant* | *float_constant* | *character_constant* | *string_constant* | **null**
interface_identifier	*identifier*
component_identifier	*identifier*
signature_identifier	*identifier*
parameter_identifier	*identifier*
identifier	*alpha* { (*alpha_num* | _) } *
boolean_constant	**true** | **false**
integer_constant	{ **-** } { *numeral* }$^{+}$
float_constant	{ **-** } { *numeral* }$^{+}$ { (**,** | **.**) { *numeral* }$^{+}$ } { (**e** | **E**) { **+** | **-** } { *numeral* }$^{+}$ }
character_constant	´ *character* ´
string_constant	**"** { *character* } * **"**
character	*numeral* | *alpha* | *special*
numeral	**1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **0**
alpha	**a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z** | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z**
special	**!** | **"** | **#** | **$** | **%** | **&** | **'** | **(** | **)** | ***** | **+** | **,** | **-** | **.** | **/** | **:** | **;** | **<** | **=** | **>** | **?** | **`** | **{** | **|** | **}** | **~**

C ORL Syntax

The Object Representation Language (ORL) has the following syntax:

representation	*rebuild* **|** *reinitialization*
rebuild	{ *interface_identifier object_migration_identifier* }*
reinitalization	{ **(** *object_migration_identifier* { *component_representation* }* **)** }*
component_representation	*relative_object_identity* | *boolean_representation* | *integer_representation* | *float_representation* | *char_representation* | *string_representation* | *representation_list* | *****
relative_object_identity	**:***object_migration_identifier*
representation_list	**(** *{component_representation }*+ **)**
object_migration_identifier	{ *numeral* }+
interface_identifier	*identifier*
identifier	*alpha* { (*alpha_num* | _) } *
boolean_representation	**true** | **false**
integer_representation	{ **-** } { *numeral* }+
float_representation	{ **-** } { *numeral* }+ { (**,** | **.**) { *numeral* }+ } { (**e** | **E**) { **+** | **-** } { *numeral* }+ }
character_representation	´ *character* ´
string_representation	**"** { *character* } * **"**
character	*numeral* | *alpha* | *special*
numeral	**1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **0**
alpha	**a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z** | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z**
special	**!** | **"** | **#** | **$** | **%** | **&** | **'** | **(** | **)** | ***** | **+** | **,** | **-** | **.** | **/** | **:** | **;** | **<** | **=** | **>** | **?** | **`** | **{** | **|** | **}** | **~**

D Standard Interfaces

The following standard Interfaces are defined for the HLM migration mechanism:

```
Interface OM_Object {
     OM_Boolean identical(OM_Object o);
     OM_String interface_name();
     OM_Boolean isa_migrateable();
};

interface OM_Boolean {
     OM_Boolean not();
     OM_Boolean and(OM_Boolean b);
     OM_Boolean or(OM_Boolean b);
     OM_Boolean xor(OM_Boolean b);
};

interface OM_Integer {
     OM_Boolean is_equal(OM_Integer i);
     OM_Boolean is_less(OM_Integer i);
     OM_Boolean is_less_equal(OM_Integer i);
     OM_Boolean is_greater(OM_Integer i);
     OM_Boolean is_greater_equal(OM_Integer i);
     OM_Integer plus(OM_Integer i);
     OM_Integer minus(OM_Integer i);
     OM_Integer times(OM_Integer i);
     OM_Float divide_by(OM_Integer i);
     OM_Float toFloat();
};

interface OM_Float {
     OM_Boolean is_equal(OM_Float f);
     OM_Boolean is_less(OM_Float f);
     OM_Boolean is_less_equal(OM_Float f);
     OM_Boolean is_greater(OM_Float f);
     OM_Boolean is_greater_equal(OM_Float f);
     OM_Float plus(OM_Float f);
     OM_Float minus(OM_Float f);
     OM_Float times(OM_Float f);
     OM_Float divide_by(OM_Float f);
     OM_Integer toInteger();
};

interface OM_Character : OM_Object {
     OM_Boolean set_this(OM_Character om_c);
     OM_Character get_this();
     OM_Character set_eof();
     OM_Boolean is_equal(OM_Character om_c);
     OM_Boolean is_less(OM_Character om_c);
     OM_Boolean is_less_equal(OM_Character om_c);
     OM_Boolean is_greater(OM_Character om_c);
     OM_Boolean is_greater_equal(OM_Character om_c);
     OM_Boolean is_printable();
     OM_Boolean is_nonprintable();
     OM_Boolean is_whitespace();
     OM_Boolean is_mumber();
     OM_Boolean is_alpha();
     OM_Boolean is_alpha_num();
     OM_Boolean is_id_char();
     OM_Boolean is_eof();
     OM_Boolean is_exponent();
     OM_Boolean eq_tab();
     OM_Boolean eq_lf();
     OM_Boolean eq_cr();
     OM_Boolean eq_blank();
     OM_Boolean eq_not();
```

```
        OM_Boolean eq_quotes();
        OM_Boolean eq_data();
        OM_Boolean eq_dollar();
        OM_Boolean eq_percent();
        OM_Boolean eq_and();
        OM_Boolean eq_quote();
        OM_Boolean eq_lb();
        OM_Boolean eq_rb();
        OM_Boolean eq_times();
        OM_Boolean eq_plus();
        OM_Boolean eq_comma();
        OM_Boolean eq_minus();
        OM_Boolean eq_point();
        OM_Boolean eq_slash();
        OM_Boolean eq_colon();
        OM_Boolean eq_semicolon();
        OM_Boolean eq_lt();
        OM_Boolean eq_equal();
        OM_Boolean eq_gt();
        OM_Boolean eq_questionmark();
        OM_Boolean eq_at();
        OM_Boolean eq_lsb();
        OM_Boolean eq_backslash();
        OM_Boolean eq_rsb();
        OM_Boolean eq_caret();
        OM_Boolean eq_underscore();
        OM_Boolean eq_backquote();
        OM_Boolean eq_lrb();
        OM_Boolean eq_or();
        OM_Boolean eq_rrb();
        OM_Boolean eq_tilt();
        OM_Integer asInteger();
        OM_Boolean fromInteger(OM_Integer i);
    };

    interface OM_String : OM_Object {
        OM_Booleam set_this(OM_String om_s);
        OM_String get_this();
        OM_Boolean is_empty();
        OM_Boolean is_equal(OM_String om_s);
        OM_Boolean is_less(OM_Character om_c);
        OM_Boolean is_less_equal(OM_Character om_c);
        OM_Boolean is_greater(OM_Character om_c);
        OM_Boolean is_greater_equal(OM_Character om_c);
        OM_Boolean contains(OM_Character om_c);
        OM_Boolean contains(OM_String om_s);
        OM_Boolean starts_with(OM_String om_s);
        OM_Boolean ends_with(OM_String om_s);
        OM_Boolean start_of(OM_String om_s, OM_Integer om_i);
        OM_Integer string_length();
        OM_Character at(OM_Integer i);
        OM_Boolean string_append(OM_Character om_c);
        OM_Boolean string_append(OM_String om_s);
        OM_Boolean string_append(OM_Integer om_i);
        OM_Boolean string_append(OM_Float om_f);
        OM_Boolean string_append_newline();
        OM_Boolean string_append_tabs(OM_Integer om_i);
        OM_String string_copy();
        OM_String nth_substring(OM_Integer om_i);
        OM_String string_left(OM_String om_s);
        OM_String string_right(OM_String om_s);
        OM_Character asCharacter();
        OM_Boolean fromCharacter(OM_Character c);
        OM_Integer asInteger();
```

```
        OM_Boolean fromInteger(OM_Integer i);
        OM_Float asFloat();
        OM_Boolean fromFloat(OM_Float f);
        OM_Boolean print_to(OM_Stream s);
    };
    interface OM_Set : OM_Object {
        OM_Set new_tail();
        OM_Boolean set_this(OM_Set om_s);
        OM_Set get_this();
        OM_Boolean is_empty();
        OM_Boolean contains_object(OM_Object om_o);
        OM_Object get_element();
        OM_Set get_next();
        OM_Boolean set_element(OM_Object om_o);
        OM_Boolean set_next(OM_Set s);
        OM_Boolean clear_next();
        OM_Boolean union_object(OM_Object om_o);
        OM_Boolean remove_object(OM_Object om_o);
        OM_Boolean union_set(OM_Set s);
        OM_Boolean intersection_set(OM_Set s);
        OM_Boolean difference_object(OM_Object om_o);
        OM_Boolean difference_set(OM_Set s);
        OM_Boolean print_to(OM_Stream s);
    };
    interface OM_Stream : OM_Object {
        OM_Boolean set_this(OM_Stream om_s);
        OM_Boolean set_to_standard_input();
        OM_Boolean set_to_standard_output();
        OM_Stream get_this();
        OM_Boolean readable();
        OM_Boolean writeable();
        OM_Character stream_read();
        OM_String stream_read_line();
        OM_String stream_read_until(OM_Character om_c);
        OM_String stream_read_unitl_eof();
        OM_Boolean stream_write(OM_Character om_c);
        OM_Boolean stream_write(OM_Integer i);
        OM_Boolean stream_write(OM_Float f);
        OM_Boolean stream_write(OM_String om_s);
        OM_Boolean stream_write_newline();
        OM_Boolean stream_write_tabs(OM_Integer n);
        OM_Boolean stream_flush();
        OM_Boolean stream_close();
    };
    interface OM_File : OM_Object {
        OM_Boolean set_this(OM_File om_f);
        OM_File get_this();
        OM_Boolean readable();
        OM_Boolean writeable();
        OM_Stream file_stream();
        OM_Boolean set_filename(OM_String om_s);
        OM_Boolean open_readable();
        OM_Boolean open_writeable();
        OM_Character file_read();
        OM_Boolean file_write(OM_Character om_c);
        OM_Boolean file_close();
    };
    interface OM_Directory : OM_Object {
        OM_Boolean set_this(OM_Directory om_d);
        OM_Directory get_this();
        OM_Boolean is_empty();
        OM_String get_entry();
```

```
        OM_Directory get_next();
        OM_Directory get_previous();
        OM_Boolean set_name(OM_String om_s);
        OM_Boolean set_entry(OM_String om_s);
        OM_Boolean set_next(OM_Directory om_d);
        OM_Boolean set_previous(OM_Directory om_d);
        OM_Boolean dir_open();
    };

    interface OM_Socket : OM_Object {
        OM_Boolean set_this(OM_Socket om_s);
        OM_Socket get_this();
        OM_Stream socket_stream();
        OM_Boolean socket_open(OM_String om_host, OM_Integer port);
        OM_Character socket_read();
        OM_Boolean socket_write(OM_Character c);
        OM_Boolean socket_close();
    };

    interface OM_ServerSocket : OM_Object {
        OM_Boolean set_this(OM_ServerSocket om_s);
        OM_ServerSocket get_this();
        OM_Stream socket_stream();
        OM_Boolean socket_listen_at_port(OM_Integer port);
        OM_Character socket_read();
        OM_Boolean socket_write(OM_Character om_c);
        OM_Boolean socket_close();
    };

    interface OM_Environment : OM_Object {
        OM_Boolean environment_initialize();
        OM_Stream terminal_stream();
        OM_Boolean load_interface(OM_String in);
        OM_Boolean generate_interface(OM_Interface i,
                            OM_String bd, OM_Integer l);
        OM_Boolean implement_interface(OM_Interface i, OM_String om_s);
        OM_Object instanciate(OM_String in);
        OM_Integer random_integer(OM_Integer limit);
    };

    interface OM_Porter : OM_Object {
        OM_Boolean set_environment(OM_Environment e);
        OM_Environment get_environment();
        OM_Boolean set_base_dir(OM_String bd);
        OM_String get_base_dir();
        OM_MigrateableSet get_migrated();
        OM_Boolean set_owner(OM_Owner o);
        OM_Owner get_owner();
        OM_Integer new_migration_id();
        OM_Boolean supported_interfaces(OM_String istr);
        OM_Boolean process_migration(OM_InterfaceSet ip,
                            OM_InterfaceSet si,
                            OM_InterfaceSet im);
        OM_Boolean process_interface(OM_Interface i,
                            OM_InterfaceSet ip,
                            OM_InterfaceSet si,
                            OM_InterfaceSet im);
        OM_Boolean initialize_representation(OM_String r_str,
                            OM_MigrateableSet ms);
        OM_Boolean collect_representations(OM_String r_str,
                            OM_MigrateableSet ms);
        OM_Interface resolve_interface_name(OM_String in);
        OM_Migrateable lookup_by_relative_identity(OM_Integer i);
        OM_Boolean implement(OM_String s);
        OM_Boolean represent(OM_String s, OM_MigrateableSet);
        OM_Boolean base_interface(OM_String om_s);
```

```
        OM_Boolean migrate_local(OM_Migrateable m, OM_Owner lo,
                            OM_Owner ro);
        OM_Boolean migrate_via_network(OM_Migrateable m,
                            OM_Owner o,
                            OM_String host,
                            OM_Integer port,);
        OM_Boolean handle_migration_via_network(OM_Owner o,
                            OM_Integer port);
        OM_Boolean migrate_via_filesystem(OM_Migrateable m,
                            OM_Owner o,
                            OM_String dir);
        OM_Boolean handle_migration_via_filesystem(OM_Owner o,
                            OM_String dir);
        OM_Boolean write_control_file(OM_String filename);
        OM_Boolean wait_for_control_file(OM_String directory,
                            OM_String filename);
        OM_Boolean porter_commit_migration();
        OM_Boolean porter_abort_migration();
    };

    interface OM_Owner : OM_Object {
        OM_Boolean set_porter(OM_Porter p);
        OM_Porter get_porter();
        OM_Boolean register_migrateable(OM_Migrateable m);
        OM_Boolean ready_to_let_component_migrate(OM_Integer mid,
                            OM_Migrateable m);
        OM_Boolean deactivate(OM_Migrateable m);
        OM_Boolean activate(OM_Migrateable m);
        OM_Boolean reactivate(OM_Migrateable m);
    };

    interface OM_Migrateable : OM_Owner {
        OM_Boolean set_relative_identity(OM_Integer i);
        OM_Integer relative_identity();
        OM_Boolean register_owner(OM_Owner m);
        OM_Boolean unregister_owner(OM_Owner m);
        OM_Boolean ready_to_migrate(OM_Integer mid,
                            OM_Owner o, OM_MigrateableSet ms);
        OM_Boolean ready_to_let_owner_migrate(OM_Integer mid,
                            OM_Owner o);
        OM_Boolean collect_interfaces(OM_Porter p, OM_InterfaceSet ip);
        OM_Boolean collect_migrateables(OM_MigrateableSet mm);
        OM_Boolean representation(OM_String r_str,
                            OM_MigrateableSet ms);
        OM_Boolean rebuild(GOAL_Token tl, OM_Porter p);
        OM_Boolean initialize_after_migration(OM_Porter p, OM_Owner o);
        OM_Boolean deactivate(OM_Porter p);
        OM_Boolean abort_migration(OM_Porter p);
        OM_Boolean activate(OM_Porter p);
        OM_Boolean reactivate(OM_Porter p);
        OM_Boolean release(OM_Porter p)
    };
```

The following additional interfaces (in alphabetical order) are used to implement the HLM migration mechanism but are not listed here for reasons of space:

OM_Access , OM_Assignment , OM_Component , OM_ComponentSet , OM_Do , OM_Error , OM_Expression , OM_If , OM_Index , OM_IndexExpression , OM_IndexExpressionSet , OM_IndexSet , OM_Instantiation , OM_Interface , OM_InterfaceSet , OM_Message , OM_MessageSend , OM_MessageSet , OM_MigrateableSet , OM_OwnerSet , OM_Parameter , OM_ParameterExpressionSet , OM_ParameterSet , OM_Return , OM_Scanner , OM_Signature , OM_SignatureSet , OM_Statement , OM_StatementSet , OM_SubExpression , OM_SyntaxElement , OM_SyntaxElementSet , OM_Token , OM_Variable , OM_VariableSet , OM_While , GOAL_Token

E Example

The following source code listing represents the interface definition of the `OM_Derivative` interface used in the example of chapter 3 on page 125. The `OM_Derivative` interface is dependent on the `OM_DerivativeSet` and the `OM_DerivativeShell` interfaces that are not shown here for reasons of space. The `OM_DerivativeSet` interface provides normal set operations for `OM_Derivative` objects. The `OM_Derivative Shell` provides the command line user interface and acts as an owner object of `OM_Derivative` objects.

```
//************************************************************************
// OM_Derivative.goal
// (C) 2000 Michael P. Wagner

//************************************************************************
interface OM_Derivative : OM_Migrateable {
      OM_String name;
      OM_String underlying_name;
      OM_Integer premium_price;
      OM_Integer execution_price;
      OM_Integer cap_price;
      OM_String expiration_date;
      OM_Derivative owner;
      OM_DerivativeSet sub_derivatives;
      OM_DerivativeShell dshell;

      //********************************************************************
      OM_Boolean derivative_init(OM_DerivativeShell ds)
      {
            dshell = ds;
            return true;
      };

      //********************************************************************
      OM_Boolean set_dshell(OM_DerivativeShell s)
      {
            dshell = s;
            return true;
      };

      //********************************************************************
      OM_Boolean set_name(OM_String s)
      {
            name = s;
            return true;
      };

      //********************************************************************
      OM_String get_name()
      {
            return name;
      };

      //********************************************************************
      OM_Boolean set_underlying_name(OM_String s)
      {
            underlying_name = s;
            return true;
      };

      //********************************************************************
      OM_String get_underlying_name()
      {
            return underlying_name;
```

```
};

//*****************************************************************
OM_Boolean set_premium_price(OM_Integer p)
{
      premium_price = p;
      return true;
};

//*****************************************************************
OM_Integer get_premium_price()
{
      return premium_price;
};

//*****************************************************************
OM_Boolean set_execution_price(OM_Integer ep)
{
      execution_price = ep;
      return true;
};

//*****************************************************************
OM_Integer get_execution_price()
{
      return execution_price;
};

//*****************************************************************
OM_Boolean set_cap_price(OM_Integer cp)
{
      cap_price = cp;
      return true;
};

//*****************************************************************
OM_Integer get_cap_price()
{
      return cap_price;
};

//*****************************************************************
// compute value based on purchase price
OM_Integer calc_value(OM_Integer pp)
{
      OM_Integer v;
      OM_Integer execution;
      OM_Integer cap;
      OM_Integer premium;
      OM_Integer quote;
      OM_Integer subv;
      OM_Boolean subflag ;

      execution = this.get_execution_price();
      cap = this.get_cap_price();
      premium = this.get_premium_price();
      if (underlying_name != null) {
            quote = dshell.get_quote(underlying_name);
      } else {
            quote = 0;
      };

      subflag = false;
      v = 0;
```

```
            if (sub_derivatives != null) {
                  if (sub_derivatives.is_empty () == false) {
                        subv = sub_derivatives.calc_value(pp);
                        if (pp > 0) {
                              v = subv - pp;
                        } else {
                              v = subv;
                        };
                        subflag = true;
                  };
            };
            if (subflag != true) {
                  if (cap > execution) {
                        // Call
                        if (quote < cap) {
                              if (quote > execution) {
                                    // in the money
                                    v = quote - execution - premium - pp;
                              } else {
                                    // at or out of the money
                                    v = - pp;
                              };
                        } else {
                              // execution pending
                              v = cap - execution - premium - pp;
                        };
                  } else {
                        // Put
                        if (quote > cap) {
                              if (quote < execution) {
                                    // in the money
                                    v = execution - quote - premium - pp;
                              } else {
                                    // at or out of the money
                                    v = -pp;
                              };
                        } else {
                              // execution pending
                              v = execution - cap - premium - pp;
                        };
                  };
            };
            return v;
      };

      //*************************************************************
      // list derivatives to a string
      OM_String derivative_list(OM_String ps, OM_Integer p, OM_Integer pp)
      {
            OM_String rs;
            OM_String subps;
            OM_String subrs;
            OM_Integer value;
            OM_Integer execution;
            OM_Integer cap;
            OM_Integer premium;
            OM_Integer quote;

            execution = this.get_execution_price();
            cap = this.get_cap_price();
            premium = this.get_premium_price();
            if (underlying_name != null) {
                  quote = dshell.get_quote(underlying_name);
            } else {
```

```
            quote = 0;
      };
      value = this.calc_value(pp);

      rs = new OM_String();
      if (ps != null) {
            if (ps.is_empty() == false) {
                  rs.string_append(ps);
                  rs.string_append(".");
            };
      };
      rs.string_append(p);
      rs.string_append_tabs(1);
      rs.string_append(name);
      rs.string_append_tabs(1);
      rs.string_append(underlying_name);
      rs.string_append_tabs(1);
      rs.string_append(execution);
      rs.string_append_tabs(1);
      rs.string_append(premium);
      rs.string_append_tabs(1);
      rs.string_append(cap);
      rs.string_append_tabs(1);
      rs.string_append(pp);
      rs.string_append_tabs(1);
      rs.string_append(quote);
      rs.string_append_tabs(1);
      rs.string_append(value);
      /*
      rs.string_append_tabs(1);
      if (value > 0) {
            rs.string_append("in");
      };
      if (value == 0) {
            rs.string_append("at");
      };
      if (value < 0) {
            rs.string_append("out");
      };
      */
      rs.string_append_tabs(1);
      rs.string_append(this.interface_name());
      rs.string_append_newline();
      if (sub_derivatives != null) {
            if (sub_derivatives.is_empty () == false) {
                  subps = new OM_String();
                  if (ps != null) {
                        if (ps.is_empty() == false) {
                              subps.string_append(ps);
                              subps.string_append(".");
                        };
                  };
                  subps.string_append(p);
                  subrs = sub_derivatives.derivatives_list(subps);
                  rs.string_append(subrs);
            };
      };
      return rs;
};

//*************************************************************
// ask for a manual quote
OM_Boolean update_quote_manually()
{
```

```
            OM_Integer quote;
            OM_Boolean ok;

            ok = true;
            quote = dshell.get_quote_manually(underlying_name);
            if (sub_derivatives != null) {
                  ok = sub_derivatives.update_quote_manually();
            };
            return ok;
      };

      //*****************************************************************
      // subsume a derivative
      OM_Boolean subsume(OM_Derivative subd)
      {
            OM_Boolean ok;

            ok = false;
            if (subd != null)
            {
                  if (sub_derivatives == null) {
                        sub_derivatives = new OM_DerivativeSet();
                  };
                  ok = subd.register_owner((OM_Owner)this); // register as
owner
                  if (ok == true ) {
                        ok = sub_derivatives.union_object(subd);
                  };
            };
            return ok;
      };

      //*****************************************************************
      // checks whether this is ready to migrate
      OM_Boolean ready_to_migrate(OM_Integer mid, OM_Owner o,
                                  OM_MigrateableSet ms)
      {
            OM_Migrateable m;
            OM_Boolean more;
            OM_Integer i;
            OM_DerivativeSet dse;
            OM_Derivative d;
            OM_OwnerSet ows;
            OM_Owner ow;
            OM_Boolean ok;

            ok = true;
            // object already visited ?
            if (current_migration_id == mid) {
                  // may be handled different for each owner
                  return true;
            };
            current_migration_id = mid;
            // is object alive ?
            if (stack_flag == true)
            {
                  return false;
            };
            // check whether owners agree
            if (owners != null) {
                  // may be handled different for each owner
                  ows = owners;
                  while ((ows != null) & (ok == true))
                  {
```

```
                        ow = (OM_Owner)ows.get_element();
                        if ((ow != null) & (ow != o)) {
                              ok =ow.ready_to_let_component_migrate(mid,this);
                        };
                        if (ok == true)
                        {
                              ows = (OM_OwnerSet)ows.get_next();
                        };
                  };
                  if (ok != true)
                  {
                        return false;
                  };
            };
            // are there components ?
            if (sub_derivatives != null) {
                  //migrateable components
                  dse = sub_derivatives;
                  while ((dse != null) & (ok == true))
                  {
                        d = (OM_Derivative)dse.get_element();
                        if (d != null) {
                              ok = d.ready_to_migrate(mid,this,ms);
                        };
                        if (ok == true)
                        {
                              dse = (OM_DerivativeSet)dse.get_next();
                        };
                  };
                  if (ok != true)
                  {
                        return false;
                  };
            };
            // add to migrationset
            if (ms != null) {
                  ms.union_object(this);
            } else {
                  // ms not initialized
                  return false;
            };
            return ok;
      };

      //*****************************************************************
      // check if component can be migrated
      OM_Boolean ready_to_let_component_migrate(OM_Integer mid,
                                                OM_Migrateable m)
      {
            OM_Boolean ok;

            // check mid ?
            ok = true;
            if (sub_derivatives != null)
            {
                  ok = sub_derivatives.contains_object(m);
            } else {
                  return false;
            };
            return ok;
      };

      //*****************************************************************
      // add interface of object to set
```

```
OM_Boolean collect_interfaces(OM_Porter p, OM_InterfaceSet ip)
{
      OM_String in;
      OM_Interface i;
      OM_Boolean ok;

      ok = true;
      in = this.interface_name();
      if (in != null) {
            i = p.resolve_interface_name(in);
            if (i != null) {
                  ip.union_object(i);
            } else {
                  // interface could not be resolved
                  return false;
            };
      } else {
            // interface name could not be retrieved
            return false;
      };
      return ok;
};

//*************************************************************
// representation()
// writes a representation of this to the given string
OM_Boolean representation(
      OM_String r_str,              // representation string
      OM_MigrateableSet ms
      )
{
      OM_Boolean ok;
      OM_Boolean more;
      OM_Integer i;
      OM_DerivativeSet dse;
      OM_Derivative d;
      OM_Integer rid;

      ok = true;
      r_str.string_append(relative_identity);
      r_str.string_append(" ");
      r_str.string_append("(");
      // atomic components
      r_str.string_append(name);
      r_str.string_append(" ");
      r_str.string_append(underlying_name);
      r_str.string_append(" ");
      r_str.string_append(premium_price);
      r_str.string_append(" ");
      r_str.string_append(execution_price);
      r_str.string_append(" ");
      r_str.string_append(cap_price);
      r_str.string_append(" ");
      //migrateable components
      if (sub_derivatives != null) {
            dse = sub_derivatives;
            while ((dse != null) & (ok == true))
            {
                  d = (OM_Derivative)dse.get_element();
                  if (d != null) {
                        rid = d.relative_identity();
                        r_str.string_append(":");
                        r_str.string_append(rid);
                        r_str.string_append(" ");
```

```
                        dse = (OM_DerivativeSet)dse.get_next();
                    };
                };
            };
            // end of representation
            r_str.string_append(") ");
            return ok;
};

//*****************************************************************
// rebuild()
// reconstructs the object structure from the token-list
OM_Boolean rebuild(
            GOAL_Token tl,
            OM_Porter p
            )
{
            OM_Error err;
            OM_Boolean ok;
            GOAL_Token gt;
            GOAL_Token nt;
            OM_String in;
            OM_String i_str;
            OM_Integer i;
            OM_Boolean more;
            OM_Migrateable m;
            OM_Derivative d;

            ok = true;
            // parse
            if (tl == null) {
                  return false;
            };
            gt = tl;
            if ((gt != null) & gt.eq_lb()) {
                  gt = gt.next_token();
                  // atomic components
                  // read own representation
                  if ((gt != null) & gt.eq_identifier()) {
                        i_str = gt.get_content();
                        name = i_str;
                        gt = gt.next_token();
                  } else {
                        // integer expected
                        return false;
                  };
                  if ((gt != null) & gt.eq_identifier()) {
                        i_str = gt.get_content();
                        underlying_name = i_str;
                        gt = gt.next_token();
                  } else {
                        // integer expected
                        return false;
                  };
                  if ((gt != null) & gt.eq_integer_constant()) {
                        i_str = gt.get_content();
                        i = i_str.asInteger();
                        premium_price = i;
                        gt = gt.next_token();
                  } else {
                        // integer expected
                        return false;
                  };
                  if ((gt != null) & gt.eq_integer_constant()) {
```

```
                  i_str = gt.get_content();
                  i = i_str.asInteger();
                  execution_price = i;
                  gt = gt.next_token();
            } else {
                  // integer expected
                  return false;
            };
            if ((gt != null) & gt.eq_integer_constant()) {
                  i_str = gt.get_content();
                  i = i_str.asInteger();
                  cap_price = i;
                  gt = gt.next_token();
            } else {
                  // integer expected
                  return false;
            };
            // non-atomic components
            if ((gt != null) & gt.eq_colon()) {
                  more = true;
                  while ((more == true) & (ok == true)) {
                        if ((gt != null) & gt.eq_colon()) {
                              gt = gt.next_token();
                              if ((gt != null) &
                                    gt.eq_integer_constant()) {
                                    i_str = gt.get_content();
                                    i = i_str.asInteger();
                                    // set component c
                                    m =p.lookup_by_relative_identity(i);
                                    if (m != null) {
                                          d = (OM_Derivative)m;
                                          if (sub_derivatives == null) {
                                                sub_derivatives =
                                                new OM_DerivativeSet();
                                          };
                                          ok =
                                    sub_derivatives.union_object(d);
                                          gt = gt.next_token();
                                    } else {
                                          ok = false;
                                    };
                              } else {
                                    ok = false;
                              };
                        } else {
                              more = false;
                        };
                  };
            };
            // finish
            if ((gt != null) & gt.eq_rb()) {
                  nt = gt.next_token();
                  if (nt == null) {
                        tl.clear_parse_at();
                  } else {
                        tl.set_parse_at(nt);
                  };
                  return true;
            } else{
                  // ) expected
                  return false;
            };
      } else {
            // ( expected
```

```
            return false;
      };
};

//*****************************************************************
// is send to the migrated root-object after reconstruction
// initializes the object-structure recursivly
OM_Boolean initialize_after_migration(OM_Porter p, OM_Owner o)
{
      OM_Boolean ok;
      OM_DerivativeSet dse;
      OM_Derivative d;
      OM_Integer i;
      OM_Boolean more;

      ok = true;
      if (o == null) {
            return false;
      };
      dshell = (OM_DerivativeShell)o;
      if (sub_derivatives != null) {
            dse = sub_derivatives;
            while ((dse != null) & (ok == true)) {
                  d = (OM_Derivative)dse.get_element();
                  if (d != null) {
                        ok =
            d.initialize_with_owner_and_dshell(p,this,dshell);
                        dse = (OM_DerivativeSet)dse.get_next();
                  } else {
                        ok = false;
                  };
            };
      };
      return ok;
};

//*****************************************************************
// is send to the migrated root-object after reconstruction
// initializes the object-structure recursivly
OM_Boolean initialize_with_owner_and_dshell(OM_Porter p, OM_Owner o,
                                            OM_DerivativeShell ds)
{
      OM_Boolean ok;
      OM_DerivativeSet dse;
      OM_Derivative d;
      OM_Integer i;
      OM_Boolean more;

      ok = true;
      if (o != null) {
            ok = this.register_owner(o);
            if ((ok == true) & (ds != null)) {
                  dshell = ds;
                  if (sub_derivatives != null) {
                        dse = sub_derivatives;
                        while ((dse != null) & (ok == true)) {
                              d = (OM_Derivative)dse.get_element();
                              if (d != null) {
                                    ok =
                        d.initialize_with_owner_and_dshell(p,this,ds);
                                    dse =
                                    (OM_DerivativeSet)dse.get_next();
                              } else {
                                    ok = false;
```

```
                                    };
                               };
                          };
                     };
               };
               return ok;
        };

        //*****************************************************************
        // is send to each local object that was migrated
        // connections should be cut by owners
        OM_Boolean deactivate(OM_Porter p)
        {
               OM_Boolean ok;

               ok = true;
               // deactivate local
               // deactivate owners
               if (dshell != null) {
                     dshell.deactivate_component(this);
               };
               return ok;
        };

        //*****************************************************************
        // is send to each migrated object when migration is aborted
        // cuts all connections
        OM_Boolean abort_migration(OM_Porter p)
        {
               // release local
               // release owners
               owners = null;
               return true;
        };

        //*****************************************************************
        // is send to the migrated root object to resume normal operation
        // connections from owners to migrated objects are established
        OM_Boolean activate(OM_Porter p)
        {
               OM_Boolean flag;

               flag = true;
               // activate local
               // activate owners
               if (dshell != null) {
                     dshell.activate_component(this);
               };
               return flag;
        };

        //*****************************************************************
        // send to local object when activation of migrated objects fails
        // reestablishes connections from owners to objects and
        // resumes normal operation
        OM_Boolean reactivate(OM_Porter p)
        {
               OM_Boolean flag;

               flag = true;
               // reactivate local
               // activate owners
               if (dshell != null) {
                     dshell.reactivate_component(this);
```

```
            };
            return flag;
      };

      //*****************************************************************
      // is send to local objects when
      OM_Boolean release(OM_Porter p)
      {
            // release local
            // release owners
            dshell = null;
            // clean up
            return true;
      };

};
```

F Literature

The following bibliographic references, ordered by the surname of the first author and the year of publication, are used throughout this work:

[AA+1995] Amarasinghe S. P., Anderson J. M., Lam M. S., Tseng C. W.; The SUIF compiler for scalable parallel machines; 7th SIAM Conference on Parallel Processing for Scientific Computing, February 1995

[AbC1996] Abadi Martin, Cardelli Luca; A Theory of Objects; Springer, Berlin, Germany; 1996

[AB+1995] Árabe José Nagib Cotrim, Beguelin Adam, Lowekamp Bruce, Seligman Erik, Starkey Mike; Dome: parallel programming in a heterogeneous multi-user environment; Technical Report, CMU-CS-95-137; Carnegie Mellon University; 1995

[Ach1991] Achauer Bernd; Distribution in Trellis/DOWL; TOOLS'91, Santa Barbara, 1991; Prentice Hall

[Ach1993a] Achauer Bernd; The DOWL distributed object-oriented language; Communications of the ACM, 36, 9; Pages 48-55; 1993

[Ach1993b] Achauer Bruno; Implementation of Distributed Trellis; ECOOP'93, Kaiserslautern, Germany, 26-30 July 1993; Springer

[AcS1996] Acharya Amurag, Saltz Joel; Dynamic Linking for Mobile Programs; Lecture Notes in Computer Science (LNCS), 1222; Mobile Object Systems, Linz, Austria, 8.7.1996; Pages 245-262; Springer, Berlin, Germany

[Agh1986a] Agha G.; Actors: A model of Concurrent Computation in Distributed Systems; MIT Press Cambridge; 1986

[Agh1986b] Agha G.; An Overview of Actor Languages; Sigplan Notices, 21, 10, 1986; Pages 58-67

[Agh1990] Agha Gul A.; Concurrent Object-Oriented Programming; CACM, 33, 9, Sep 1990; Pages 125-141

[AgJ1999] Agha Gul A., Jamali Madeem; Concurrent Programming for DAI; in: Multiagent Systems; Gerhard Weiss (Editor); MIT Press; 1999

[AIK2000] Abe H., Ichisugi Y., Kato K.; Implementing mobile threads in java with source code translation technique; IPJS Transactions on Programming, 41, 6, Mar 2000; Pages 29-40

[AJ+1992] Amaral Paulo, Jacquemont Christian, Jensen Peter, Lea Rodger, Mirowski Adam; Transparent object migration in COOL-2; ECOOP'92, Utrecht, Netherlands, 29 June 1992

[ALJ1991] Amaral Paulo, Lea R., Jacquemot C.; COOL-2: An object oriented support platform built above the Chorus Micro-Kernel; Workshop on Object Orientation in Operating Systems, Palo Alto, 17-18 October 1991; Pages 68-73; IEEE

[Alm1985] Almes Guy T., Black Andrew P., Lazowska Edward D., Noe Jerre D.; The Eden System: A Technical Review; Transactions on Software Engineering, 11, 1, January 1985; Pages 43-59

[ArF1989] Artsy Y., Finkel R.; Designing a proces migration facility: The Charlotte experience; Computer, September 1989; Pages 47-56; IEEE

[ARS1996] Acharya Amurag, Ranaganathan M., Saltz Joel; Sumatra: A Language for Ressource Aware Mobile Programs; Lecture Notes in Computer Science (LNCS), 1222; Mobile Object Systems, Linz, Austria, 8.7.1996; Pages 111-130; Springer, Berlin, Germany

[AvV2000] Avvenuti Marco, Vecchio Alessio; MobileRMI, a toolkit for enhancing Java Remote Method Invocation with mobility; ECOOP'2000 Workshop Mobile Object Systems, Sophia Antipolis, France, June 2000

[BaL1985] Barak A. B., Litman A.; MOS: A Multi-computer Distributed Operating System; Software Practise and Experience, Aug 1985; Pages 725-737

[BaM1995] Baumgarten Uwe, Maier Joachim; Using Mobile Objects with a PDA; ECOOP'95; 1995

[BaP1998] Baldi Mario, Picco Gian Pietro; Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications; Software Engineering'98, Kyoto, Japan; Pages 146-155; 1998

[Bau1996] Baumgarten Uwe; Personal Assistance by means of Mobile Objects; ECOOP'96 Workshop Reader, Linz, Austria, 7. 1996; Pages 370-373; dpunkt, Heidelberg, Germany

[BaW1989] Barak A., Wheeler R.; MOSIX: An Integrated Multiprocessor Unix; Usenix'89, Freb 1989

[Ba+1989] Bal E., et al.; Programming Languages for Distributed Computing Systems; Computing Surveys, 21, 3, September 1989; Pages 261-322; ACM, New York, NY, USA

[BA+1995] Bellisard Luc, Atallah Slim Ben, Kerbat Alain, Riveill Michael; Component-based Programming and Application Management with Olan; OBPDC'95, Tokyo, Japan, June1995; Pages 290-309

[BB+1991] Balter Roland, Bernadat J., Decouchant D., Duda A., Freyssinet A., Krakoviak S., Meysembourg M., Le Dot P., Nguyen H. van, Paire E., Riveill Michel, Roisin C., Pina Xavier Rousset de, Scioville R., Vandôme G.; Architecture and Implementation of Guide, an Object-Oriented Distributed System; Computing Systems, 4, 1, 1991

[BB+1995] Banâtre Michael, Belhamissi Yasimna, Issarny Valérie, Puaut Isabelle, Routeau Jean-Paul; Isatis: A Customizable Distributed Object-Based Runtime System; OBPDC'95, Tokyo, Japan, June1995

[BB+1999] Bolshakov Kirill, Borshchev Andrei, Flippoff Alex, Karpov Yuri, Roudakov Victor; Creating and Running Mobile Agents with XJ DOME; PaCT'99, St.Peterburg, Russia, September 1999; Pages 410-416

[BeH2000] Belle Werner van, Hondt Theo D'; Agent Mobility and Reification of Computational State; ECOOP'2000 Workshop Mobile Object Systems, Sophia Antipolis, France, June 2000

[BeL1993] Calico Programmers Manual; Bell Labs, Naperville, IL, USA; 1993

[Ben1987] Bennett John K.; Distributed Smalltalk: inheritance and reactiveness in distributed systems; TR-87-12-04, Dec 1987; University of Washington

[Ben1990] Bennett John K.; Experience with distributed Smalltalk; Software - Practice and Experience, 20, 2; Pages 157-180

[BeP1998] Bernardo Luis, Pinto Paulo; Scalable Service Deployment Using Mobile Agents; LNCS, 1477; MA'98, Stuttgart, Germany, September 1998; Pages 261-272

[BeR1996] Bellissard Luc, Riveill Michel; From Distributed Objects to Distributed Components: the OLAN Approach; ECOOP'96, Linz Austria, July 1996; D-Punkt

[BGP1997] Baldi Mario, Gai Silvano, Picco Gian Pietro; Exploiting Code Mobility in Decentralized and Felxible Network Management; Mobile Agents, Berlin, Germany, 7.4.1997; Pages 13-26; Springer, Berlin, Germany

[BGW1993] Barak A., Guday S., Wheeler R.G.; The MOSIX Distributed Operating Syste: Load Balancing for Unix; Lecture Notes in Computer Science, 672; Springer; 1993

[BG+1991] Bricker Allan, Gien Michael, Guillemont Marc, Lipkis Jim, Orr Doug, Rozier Marc; Architectural issues in microkernel-based operating systems: the CHORUS experience; Computer Communications, 14, 6, July 1991; Pages 347-357

[BG+1992] Boukachour Jaouad, Galinho Thierry, Itmi Mhamed, Pécuchet Jean-Pierre; An Experimental Translation from Flavors to Clos; TOOLS'92, Dortmund, Germany, April 1992; Pages 391-398; Prentice Hall

[BhC1995] Bharat Krishna, Cardelli Luca; Migratory Applications; User Interface Software and technology, 8th ACM Symposium on, November 1995; Pages 133-142

[BH+1986a] Black Andrew P., Hutchinson Norman, Jul Eric, Levy Henry; Object Structure in the Emerald System; SIGPLAN Notices, 21, 11, 1986; OOPSLA'86, Portland, Oregon, USA; Pages 78-86

[BH+1987] Black Andrew P., Hutchinson Norman, Jul Eric, Levy H., Carter L.; Distribution and Abtract Types in Emerald; Transactions on Software Engineering, 13, 1, January 1987; Pages 65-75; IEEE

[BH+1998] Bursell Michael, Hyton Richard, Donaldson Douglas, Herbert Andrew; A Mobile Object Workbench; LNCS, 1477; MA'98, Stuttgart, Germany, September 1998; Pages 136-147

[Bin2000] Binder Walter; Design and Implementation of the J-Seal2 Mobile Agent Kernel; ECOOP'2000 Workshop Mobile Object Systems, Sophia Antipolis, France, June 2000

[BKT1992] Bal Henri E., Kaashoek Frans, Tanenbaum Andrew S.; Orca: A Language for Parallel Programming of Distributed Systems; Transactions on Software Engineering, 18, 3, March 1992

[Bla1985a] Black Andrew P.; Supporting Distributed Applications: Experience with Eden; Operating System Principles, Orcas Island, Washington, USA, 1985; Pages 181-193

[BlA1989] Black Andrew P., Artsy Y.; Implementing Location Independent Invocation; Distributed Computing Systems, Newport Beach, 1989; Pages 550-559

[Bla1994] Blaschek Günther; Object-Oriented Programming with Prototypes; Springer, Berlin, FRG; 1994

[Bor1986] Borning Alan H.; Classes versus prototypes in object-oriented languages; Fall JCC'86, Dallas, TX, USA, Nov 19986; Pages 36-40

[Bor1990] Borghoff Uwe M.; Dynamische Dateiallokation innerhalb eines volltransparenten verteilten Dateisystems; Dissertation; Technische Universität München, Institut für Informatik, München, FRG; 1990

[Bor1992] Borghoff Uwe; Catalog of Distributed File/Operating Systems; Springer, Berlin; 1992

[Bor1993] Borghoff Uwe M.; Möglicher Einsatz von Votierungsverfahren zur Nebenläufigkeitskontrolle in synchronen Groupware-Systemen; Habilitation; technische Universität München, Institut für Informatik, München, FRG; 1993

[BoS1995] Borghoff Uwe M., Schlichter Johann H.; Rechnergestützte Gruppenarbeit. Eine Einführung in verteilte Anwendungen; Springer, Germany; 1995

[BoW1995] Böszörményi László, Weich Carsten; Programmieren in Modula-3; Springer, Berlin, Germany; 1995

[BrG1993] Bracha Gilad, Griswold David; Strongtalk: Typechecking Smalltalk in a Production Environment; SIGPLAN Notices, 28, 10, 1993; Pages 215-230

[BrM1993] Brandt Soren, Madsen Ole Lehrmann; Object-Oriented Distributed Programming in Beta; Lecture Notes in Computer Science, 791; ECOOP'93, Kaiserslautern, Germany, July 1993; Pages 185-212; Springer, Berlin, Germany

[CaG1999] Cardelli Kuca, Gordon Andrew D.; Types for Mobile Ambients; POPL'99, San Antonio, TX, USA, 20.1.1999; Pages 79-92

[Car1994] Luca Cardelli; Obliq : a Language with distributed scope; DEC SRC Technical Report, 122; 1994

[Car1995a] Luca Cardelli; A Language with distributed scope; Computing Systems, 8, 1; Pages 27-55; 1995

[Car1996a] Cardelli Luca; Mobile Computation; Lecture Notes in Computer Science (LNCS), 1222; Mobile Object Systems, Linz, Austria, 8.7.1996; Pages 3-6; Springer, Berlin, Germany

[Car1996b] Cardelli L.; Global Computations; Computing Surveys, 28, 4, Dec 1996; Pages 27-59

[CaS1991] Carr Robert, Shafer Dan; The Power of Penpoint; Addison-Wesley, Reading MA, USA; 1991

[CA+1989b] Chase Jeffrey S., Amador Franz G., Lazowska Edward D., Levy Henry M., Littefield Richard J.; The Amber System: Parallel Programming on a Network of Multiprocessors; Operating System Review, 23, 5; SIGOPS, Litchfield Park, Arizona, USA, December 1989; Pages 147-158

[CB+1998] Christochoides Nikos, Barker Kevin, Nave Demian, Hawblitzel Chris; The Mobile Object Layer: a run-time Substrate for Mobile Adaptive Computations; Lecture Notes in Computer Science, 1505; Computing in object-oriented and parallel Systems, Santa Fe, NM, USA, Decmeber 1998

[CF+1994] Chevalier P. Y., Freyssinet A., Hagimont D., Krakowiak S., Lacourte S., Mossiere J., Rina X. Rousset de; Persistent shared object support in the Guide system: Evaluation and related work; OOPSLA'94; Pages 129-144; 1994

[CG+1995] Chess David M., Grosof B., Harrison C., Levine D., Parris C., Tsudik G.; Itinerant Agents for mobile Computing; IEEE Personal Communications Magazine, October 1995

[CG+1996a] Cugola Gianpaolo, Ghezzi Carlo, Picco Gian Pietro, Vigna Giovanni; A Characterization of Mobility and State Distribution in Mobile Code Languages; ECOOP'96 Workshop Reader, Linz, Austria, 7. 1996; Pages 309-318; dpunkt, Heidelberg, Germany

[CG+1996b] Cugola Gianpaolo, Ghezzi Carlo, Picco Gian Pietro, Vigna Giovanni; Analyzing Mobile Code Languages; Lecture Notes in Computer Science; Mobile Object Systems'96, Linz; Springer; 1996

[Cha1992] Chambers Craig; Object-Oriented Multi-Methods in Cecil; ECOOP'92, Utrecht, Netherlands, July 1992

[Cha1993] Craig Chambers; The Cecil Language; TR 93-03-05; University of Washington, USA; 1993

[Cha1999] Chatterjee Saurav; Dynamic Application Structuring on Heterogeneous, Distributed Systems; IPPS/SPDP'99, San Juan, Puerto Rico, USA, April 1999; Pages 442-453

[ChC1991] Chin R. S., Chanson S. T.; Distributed Object-Based Programming Systems; Computing Surveys, 23, 1, March 1991; Pages 91-124; ACM

[ChC1997] Chang Daniel T., Covaci Stefan; The OMG Mobile Agent Facility: A Submission; Mobile Agents, Berlin, Germany, 7.4.1997; Pages 98-110; Springer, Berlin, Germany

[Che1988] Cheriton D. R.; The V Distributed System; Communications of the ACM, 31, 3, 1988; Pages 314-333

[Che1998] Chess David M.; Security Issues in Mobile Code Systems; LNCS 1419; Pages 1-14; 1998

[CHK1996] Chess David M., Harrison Colin , Kershenbaum Aaron; Mobile Agents: Are They a Good Idea ?; Lecture Notes in Computer Science (LNCS), 1222; Mobile Object Systems, Linz, Austria, 8.7.1996; Pages 25-45; Springer, Berlin, Germany

[ChU1991] Chambers Craig, Ungar David; Making Pure Object-Oriented Languages Practical; OOPSLA'91, October 1991

[CH+1991] Cahill Vinny, Horn Chris, Starovic Gradimir, Lea Rodger, Sousa Pedro; Supporting Object Oriented Languages on the Commandos Platform; CS/TR-91-56; ESPRIT'91, Brusseles, 25-29 November 1991

[CH+1993] Cahill Vinny, Harris Neville R., Balter Roland, Pina Xavier Rousset de; The COMANDOS Distributed Application Plattform; ESPRIT, 2071; Springer; 1993

[CiR1997] Ciancarini Paolo, Rossi Davide; Jada: Coordination and Communication for Java Agents; Lecture Notes in Computer Science (LNCS); Mobile Object Systems (MOS'96), Linz, Austria, July 1996

[CI+1993] Campbell Roy H., Islam Nayeem, Raila David, Madany Peter; Designing and Implementing CHOICES: an object-oriented System in C++; Communications of the ACM, 36, 9, September 1993; ACM

[CKW1996] Ciupke Oliver , Kottmann Dietmar A. , Walter Hans-Dirk; Object Migration in Non-Monolithic Distributed Applications; ICDCS'96; Pages 529-536; 1996

[CKW1998] Chuang, Tyng-Ruey, Kuo Y. S:, Wang Chein-Min; Non-Intrusive Object Introspection in C++: Architecture and Application; ICSE'98, Kyoto, Japan, 19.4.1998; Pages 312-321

[CL+1992] Capobianchi Riccardo, Lanusse Agnes, Guerraoui Rachid, Roux Pierre; Active Objects on Parallel Machines; TOOLS'92, Dortmund, Germany, April 1992; Pages 207-216

[CoN1991] Cox Brad J., Novobilski Andrew J.; Object Oriented Programming - An Evolutionary Approach; Addison-Wesley, Reading MA, USA; 1991

[CoP1995] Cotter Sean, Potel Mike; Inside Taligent Technology; Addison-Wesley, Reading MA, USA; 1995

[CoS1987] Cox Brad, Schmucker Kurt; Producer - A tool for translating Smalltalk-80 to Objective C; SIGPLAN Notices, 22, 12, 1987; OOPSLA'87; Pages 423-429

[CPV1997] Carzaniga Antonio, Picco Gian Pietro, Vigna Giovanni; Designing Distributed Applications using Mobile Code Paradigma; International Conference on Software Engineering ICSE'97, Boston, May 1997; Pages 22-32; ACM

[Cra1996] Craighill Nancy; Openstep for Enterprises; John Wiley & Sons;1996

[CS+1998] Chen Wen-Shyen E., Su S. T., Lien Yao-Nan, Shu H.T., Liu Huiling; Mobility and Management Support for Mobile Agents; Autonomous Agents, Minneapolis, MN, USA, 9.5.1998; Pages 451-452

[CT+2000] Coninx Tim, Truyen Addy, Vanhaute Bart, Berbers Yolande, Joosen Wouter, Verbaeten Pierre; On the use of Threads in Mobile Object Systems; ECOOP'2000 Workshop Mobile Object Systems, Sophia Antipolis, France, June 2000

[DaM1970] Dahl O.-J., Myrhaug B.; SIMULA Common Base Language; S-22; Norwegian Computer Center; 1970

[Dec1986] Decouchant Dominique; Design of a Distributed Object Manager for the Smalltalk-80 System; OOPSLA'86, Portland, Oregon, USA, October 1986; Pages 444-452

[DiR1998] Dimitrov Bozhidar, Rego Vladimir; Arachne: A portable threads system supporting migrant threads on heterogeneous network farms; Transactions on Parallel and Distributed Systems, 9, 5, May 1998; Pages 459-469s; IEEE

[DLA1988] Dasgupta Partha, LeBlanc Richard J., Applebe William F.; The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work; Distributed Computing Systems'88, San Jose, June 1988; Pages 29; IEEE

[DL+1991] Dasgupta P., LeBlanc R.J., Ahamad M., Ramachandran U.; The Clouds distributed operating system; Computer, 24, 1; Pages 33-44; IEEE

[DNX1992] Dollimore Jean, Nascimento Claudio, Xu Wang; Fine Grained Object Migration; International Workshop on Distributed Object Management IWDOM'92, Edmonton, CD, 19.8.1992; Pages 182-186; Morgan Kaufmann

[Doe1996] Dömel Peter; Mobile Telescript Agents and the Web; COMPCON'96; Pages 52-57; IEEE; 1996

[DoM1992] Douglis Fred, Marsh Brian; The Workstation as a Waystation: Integrating Mobility into Computing Environmnets; 3rd Workshop on Workstation Operating Systems, April 1992; IEEE

[DoO1987] Douglis Frederick, Ousterhout J.; Process migration in the Sprite operating system; DCS'87; Pages 18-25; 1987

[DoO1990] Douglis Frederick, Ousterhout J.; Transparent process migration: Design Alternatives and the Sprite Implementation; Software - Practice and Experience, 21, 8, August 1991; Pages 757-785

[Dou1990a] Douglis Frederick; Transparent process migration in the sprite operating system; Computer Science Division (EECS), Univ. of Calif. Berkeley, USA; 1990

[DO+1991] Douglis Frederick, Osterhout J. K., Kaashoeck M. F., Tanenbaum A. S.; A comparison of two distributed systems: Amoeba and Sprite; Computing Systems, 4, 3, December 1991; Pages 353-384

[DRS1989a] Duach Brent F., Rutherford Robert M., Shub Charles M.; Process-originated migration in a heterogeneous environment; Computer Science Conference, Louisville, KY, USA, February 1989; Pages 98-102

[Dug1997b] Dömel Peter; Interaction of Java and Telescript Agents; Lecture Notes in Computer Science (LNCS); Mobile Object Systems (MOS'96), Linz, Austria, July 1996

[EG+1988] Eberle H., Geihs K., Schill Alexander B., Schoener H., Schmutz H.; Generic Support for Distributed Processing in heterogeneous Networks; HECTOR Proceedings Vol. 2, Berlin; Springer; 1988

[EKO1994] Engeler D. R:, Kaashoek M.F., O'Toole J. W. Jr.; The Operating System as a Secure Programmable Machine; SIGOPS'94, September 1994; ACM

[ELZ1988] Eager D. L., Lazowska E.D., Zahorjan John; The Limited Performance Bennefits of Migrating Active Processes for Load Sharing; Measurement and Modeling of Computer Systems, Sigmetrics 1988, Santa Fe, New Mexico, USA, May 1988; Pages 63-72

[Fal1987] Falcone F.; A Programmable Interface Language for Heterogeneous Systems; Transactions of Computer Systems, 5, 4, November 1987; Pages 330-351; ACM

[FB+1997] Fukuda Munehiro, Bic Lubomir F., Dillencourt Michael B., Merchant Fehmina; Messages versus Messengers in Distributed Programming; DCS'97; Pages 347-354; IEEE; 1997

[FeD1999] Fezzani Djamel, Desbiens Jocelyn; Epidaure: A Java Distributed Tool for Building DAI Applications; EuroPar'99, Toulouse, France, August 1999

[FJ+1993] Finke S., Jahn P., Langmack O., Löhr K.-P., Piens I., Wolff Thomas; Distribution and Inheritance in the HERON Approach to Heterogeneous Computing; DCS'93; Pages 399-408; IEEE, USA; 1993

[FoZ1994] Forman George H., Zahorjan John; The Challenges of Mobile Computing; Computer, 27, 4, 4. 1994; Pages 38-47

[FPV1998] Fuggetta Alfonso, Picco Gian Pietro, Vigna Giovanni; Understanding Code Mobility; IEEE Transactions on Software Engineering, 24, 5, 5. May 1998

[Fre1991] Freeman D.; Experience Building a Process Migration Subsstem for Unix; Winter USENIX, Jan 1991; Pages 349-355

[FrH1991] Freisleben Bernd , Heck Andreas; Towards Dynamically Adaptive Operating Systems; GI/ITG Fachtagung Kommunikation in verteilten Systemen, Mannheim, 1.2.1991; Springer, Berlin, Germany

[Fri1984] Firtzson Peter; Preliminary Experience from the DICE system a Distributed Incremental Dompiling Environment; Pages 113-123; ACM; 1984

[Fuc1995] Fuchs Matthew; Dreme: for Life in the Net; Dissertation; New York University; 1995

[Fue1998] Fünfrocken, Stefan; Transparent Migration of Java-Based Mobile Agents; LNCS, 1477; MA'98, Stuttgart, Germany, September 1998; Pages 26-37

[GaY1993] Gao Yaoqing Gao, Yuen Chung Kwong; A Survey of Implementations of Concurrent, Paralle and Distributed Smalltalk; SIGPLAN Notices, 28, 9; Pages 29-35; ACM, USA

[GB+1999] Gottbrath Chris, Bailin Jeremy, meakin Casey, Thompson Todd, Charfman J.J.; The Effects of Moores's Law and Slacking on Large Computations; http://agave.as.arizona.edu/~chrisg/mooreslaw/Paper.html

[GeL1994] Gerns Arnd, Lie J. S.; Diomedes - Objectmigrationsstrategien für verteilte Umgebungen; Hildesheier Informatikberichte, 94, 27, 1994

[Ger1998] Gerns Arnd; Entwicklung und Bewertung von Objektmigrationsstrategien für verteilte Umgebungen; Universtität Oldenburg

[GHJ1994] Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John; Design Patterns - Elements of reusable Object-Oriented Software; Addison-Wesley, Reading MA, USA; 1994

[GhV1997] Ghezzi Carlo, Vigna Giovanni; Mobile Code Paradigms and Technologies: A Case Study; Mobile Agents, Berlin, Germany, 7.4.1997; Pages 40-49; Springer, Berlin, Germany

[GJS1996] Gosling James, Joy Bill, Steele Guy; The Java Language Specification; Addison-Wesley; 1996

[Gla1998] Glass Graham; ObjectSpace Voyager - The Agent ORB for Java; LNCS, 1368; WWCA'98, Tsukuba, Japan, March 1998; Pages 38-55

[Goe1998] Görl Harald; Realisierung der Sicherheit von Agenten in unterschiedlichen Agentensystemen; Diplomarbeit; Technische Universität München; 1998

[GoR1983] Golberg Adele, Robson David; Smalltalk-80, The Language and its Implementation; Addison-Wesley, USA; 1983

[Gos1991] Goscinski A.; Distributed Operating Systems: The Logical Design; Addison-Wesley; 1991

[Gsc2000] Gschwind Thomas; Comparing Object Oriented Mobile Agent Systems; ECOOP'2000 Workshop Mobile Object Systems, Sophia Antipolis, France, June 2000

[GS+1998] Geier M., Steckermeier M., Becker U., Hauck F. J., Meier E., Rastofer U.; Support for Mobility and Replication in the AspectIX Architecture; LNCS, 1543; ECOOP'98 Workshop Code Analysis and Tools, Brussels, Belgium, July 1998; Pages 325-326

[HaB1998] Hayzelden A. L. G., Bigham J.; Heterogeneous Multi-Agent Architecture for ATM Virtual Path Network Resource Cnofiguration; LNAI / LNCS, 1437; IATA'98, Paris, France, July 1998; Pages 45-59

[HäR1997] Härtig Hermann, Reuther Lars; Encapsulating Mobile Objects; DCS'97; Pages 355-362; IEEE; 1997

[Hau1998] Haustein Tobias; Objectmigration für CORBA-basierte Verteilungsplattformen; RWTH Aachen; 1998

[HBS1973] Hewitt C., Bishop P., Steiger R.; A universal modular actor formalism for artificial intelligence; IJCAI; Pages 235-245; 1973

[HCS1997] Hurst Leon , Cunningham Padraig, Somers Fergal; Mobile Agents Smart Messages; Mobile Agents, Berlin, Germany, 7.4.1997; Pages 111-122; Springer, Berlin, Germany

[HCU1992] Hölzle Urs, Chambers Craig, Ungar David; Debugging Optimized Code with Dynamic Deoptimization; PLDI'92, San Francisco, CA, USA, June 1992

[Hew1980] Hewitt C.; The Apiary network architecture for knowledgeable systems; Lisp Conference 1980, Palo Alto, Ca, USA, August 1980; Stanford University, 1980

[HMA1990] Habert Sabine, Mosseri Laurence, Abrossimov Vadim; COOL : Kernel Support for Object-Oriented Environments; SIGPLAN Notices, 25, October 1990; OOPSLA'90, Ottawa, Canada, 1990; Pages 269-277

[HM+1996] Hartman J., Manber U., Peterson L., Proebsting T.; A new paradigm for networked systems; Technical Report, 96, 11, June 1996; University of Arizona

[Hoa1974] Hoare C. A. R.; Monitors: an operating System Structuring Concept; Communications of the ACM, 10, 1974; ACM

[Hoa1978] Hoare C. A. R.; Communicating Sequential Processes; Communications of the ACM, 8, 1978; ACM

[HoB1997] Holder Ophir, Ben-Shaul Israel; A Reflective Model for Mobile Software Objects; DCS'97; Pages 339-346; IEEE; 1997

[HoC1991] Horn Chris, Cahill Vinny; Supporting distributed Applications in the Amadeus Environment; Computer Communications, 14, 6, July 1991; Pages 358-365

[Hof1996] Hof Markus; Partially Distributed Objects; ECOOP'96, Linz, Austria, July 1996; Pages 211-215; D-Punkt

[HoS1988] Hollander Y., Silbermann G. M.; A Mechanism for the Migration of Tasks in Heterogenous Distributed Processing Systems; International Conference on Parallel Processing and Applications, L'Aquila, Italy, September 1987; Pages 93-98; North Holland

[HR+1998] Haridi Seif, Roy Peter van, Brand Per, Schulte Christian; Programming languages for distributed applications; New Generating Computing, 16, 3, May 1998

[HST1996] Hauck Franz J. , Steen Maarten van , Tanenbaum Andrew S.; A Location Service for Worldwide Distributed Objects; ECOOP'96 Workshop Reader, Linz, Austria, 7. 1996; Pages 384-388; dpunkt, Heidelberg, Germany

[Hut1988] Hutchinson Norman C.; Emerald: an object-oriented language for distributed Programming; Ph.D. Thesis; Universitz of Washington, Seattle, USA; 1988

[IBM1993] SOMobjects Developer Toolkit; IBM; 1993

[IK+1997] Ingalls Dan, Kaehler Ted, Maloney John , Wallace John, Kay Alan; Back to the Future - The Story of Squeak, A Practical Smalltalk Written in Itself; OOPSLA'97; 1997

[IoD1998] Ioannidis Sotiris, Dwarkadas Sandhya; Compiler and Run-Time Support for Adaptive Load Balancing in Software Distributed Shared Memory Systems; Lecutre Notes in Computer Science, 1511; Language, Compiler and Runtime Systems LCR'98, Pittsburgh, PA, USA, May 1998; Pages 107-122

[JJC1995] Jaquemont Christian, Jensen Peter Strarup, Carrez Stephane; CHORUS Object Oriented Technology; LNCS, 1107; OBPDC'95, Tokyo, Japan, June 1995; Pages 187-204

[JL+1988] Jul Eric, Levy H., Hutchinson Norman, Black Andrew P.; Fine-Grained Mobility in the Emerald System; Transactions on Computing Systems, 6, 1, February 1988; Pages 109-133; ACM

[Joh1998] Johansen Dag; Mobile Agent Applicability; LNCS, 1477; MA'98, Stuttgart, Germany, September 1998; Pages 80-98

[JRS1994] Johansen D., Renesse R. van, Schneider F B.; Operating system support for mobile agents; Fifth IEEE Workshop on Hot Topics in Operating Systems, May 1994

[JR+1995] Johansen Dag, Renesse Robert van, Schneider Fred B.; An Introduction to the TACOMA Distributed System; CS TR 95-23, June 1995; University Tromso

[JuH1994] Julienne Astrid M., Holz Brian; ToolTalk and 0pen Protocols : Inter-Application Communication; SunSoft Press, USA; 1994

[Jul1993] Jul Eric; Separation of Distribution and Objects; Lecture Notes in Computer Science; ECOOP'93, Kaiserslautern, Germany, July 1993

[Kat1997] Kato Kazuhiko; Safe and Secure Execution Mechanism for Mobile Objects; Lecture Notes in Computer Science (LNCS); Mobile Object Systems (MOS'96), Linz, Austria, July 1996

[KaT1998] Karnik Neeran M., Tripathi Anand R.; Design issues in mobile-agent progamming systems; Concurrency; Pages 52-61; IEEE; 1998

[Kee1989] Keene Sonya E.; Object-Oriented Programming in Common Lisp; Addison-Wesley, Reading MA, USA; 1989

[KeH1998] Keller Ralph, Hölzle Urs; Binary Component Adaptation; LNCS, 1445; ECOOP'98, Brussels, Belgium, July 1998; Pages 1-12

[KKM1996] Kono K., Kato K., Masuda T.; An implementation method of migrateable distributed object using a PPC technique integrated with virtual memory; LNCS, 1098; ECOOP'96; 1996

[KK+1994] Keller L., Kilgr C., Kottmann D., Moerkotte G., Schill Alexander B., Walter Hans-Dirk, Zachmann A.; Aktive und mobile Objekte als Modellierungskonzept für dezentrale Ingeniueranwendungen; Softwaretechnik in Automatisierung und Kommunikation STAK'94, Ilmenau, 1.3.1994

[KM+1999a] Kato K., Matsubara K., Someya Y., Itabashi K., Moriyama Y.; PLANET: An open mobile object system for open network; ASA/MA'99, Palm Springs, CA, USA, 1999; IEEE

[KM+1999b] Kato K., Matsubara K., Someya Y., Itabashi K., Moriyama Y.; PLANET: An open mobile object system for open network; PDSIA'99, 1999

[KM+2000] Kato Kazuhiko, Matsubara Katsuya, Yuichi Someya, Itabashi Kazumasa, Moriyama Yutaka, Yoshia Masatoshi; Mobile Substrate: Experiences of Middleware-Layer Object Mobility; ECOOP'2000 Workshop Mobile Object Systems, Sophia Antipolis, France, June 2000

[Kna1996] Knabe Frederick; An Overview of Mobile Agent Programming; Lecture Notes in Computer Science, 1192; Analysis and Verification of Multiple-Agent Languages (LOMAPS), Stockholm, Sweden, June 1996; Springer

[Koc1997] Koch Michael; Unterstützung kooperativer Dokumentenbearbeitung in Weitverkehrsnetzen; Dissertation; Technische Universität München, Institut für Informatik, München, FRG; 1997

[Kov1996] Kovács E.; Advanced Trading Servce Through Mobile Agents; Trends in Distributed Systems '96, Aachen, Germany, 1996; Pages 112-124

[KRB1991] Kiczales Gregor, Rivieres Jim de, Bobrow Daniel G.; The Art of the Metaobject Protocol; MIT Press, Cambridge MA, USA; 1991

[KRR1998] Kovacs Ernö, Röhrle Klaus, Reich Matthias; Integrating Mobile Agenst into the Mobile Middleware; LNCS, 1477; MA'98, Stuttgart, Germany, September 1998; Pages 99-111

[KrS1999] Krone Oliver, Schubiger Simon; WebRes: Topwards a Web Operating System; 11. ITG/GI-Fachtagung Kommunikation in Verteilten Systemen (KiVS), Darmstadt, März 1999; Pages 418-429

[KT+1996] Kazuhiko Kato, Kunihiko Toumura, Katsuya Matsubara, Susumu Aikawa, Jun Yoshida, Kenji Kono, Kenjiro Taura, Tatsurou Sekiguchi; Protected and Secure Mobile Object Computing in PLANET; ECOOP'96 Workshop Reader, Linz, Austria, 7. 1996; Pages 319-326; dpunkt, Heidelberg, Germany

[KüP1998] Küpper Axel, Park Anthony S.; Stationary versus Mobile User Agents in Future Mobile Telecommunication Networks; LNCS, 1477; MA'98, Stuttgart, Germany, September 1998; Pages 112-123

[Lan1998] Lange Danny; Mobile Objects and Mobile Agents: The Future of Distributed Computing ?; LNCS, 1445; ECOOP'98, Brussels, Belgium, July 1998; Pages 1-12

[LC+1987] Liskov Barbara, Curtis Dorothy, Johnson Paul, Scheifler Robert; Implementation of Argus; Operating Systems Principles, Austin, Texas, USA, November 1987; Pages 111-122; ACM, New York, USA

[LeW1991] Lea Rodger, Weightman James; Supporting object oriented languages in a distributed environment: The COOL approach; TOOLS'91 USA, Santa Barbara, July 1991

[LG+1997] Liberman B., Griffel F., Merz M., Lamersdorf W.; Java-Based Mobile Agents - How to Migrate, Persist and Interact on Electronic Service Markets; Mobile Agents, Berlin, Germany, 7.4.1997; Pages 27-38; Springer, Berlin, Germany

[LHW1991] Lee Y. S., Huang J. H., Wang F. J.; A Distributed Smalltalk Based on Process-Object Model; CommpSac'91, Tokyo, Japan, 11.9.1991; IEEE

[Lie1986] Liebermann Henry; Using Prototypical Objects to implement Shared Behavior in Object Oriented Systems; OOPLSA'96, Portland, OR, USA, Oct 1986; Pages 214-223

[LiH1999] Linnhoff-Popien C., Haustein Tobias; Das Plug-in-Modell zur Realisierung mobiler CORBA-Objekte; Informatik aktuell; 11. ITG/GI-Fachtagung Kommunikation in Verteilten Systemen (KiVS), Darmstadt, März 1999; Pages 196-221

[Lin1998] Linnhoff-Popien C.; CORBA - Kommunikation und Management; Springer; 1998

[Lis1988a] Liskov Barbara; Distributed Programming in Argus; Communications of the ACM, 31, 3, March 1988; Pages 300-312

[LJP1993] Lea Rodger, Jacquemont Christian, Pillevesse Eric; COOL: System support for distributed object-oriented programming; Communications of the ACM, 36, 9, September 1993; Pages 37-46

[LoF1997] Löwis Martin von, Fischbeck Nils; Das Python-Buch; Addison-Wesley, Bonn, Germany; 1997

[LSW1995] Lucco S., Sharp O., Whabe R.; Omniware: A universal substrate for web programming; World Wide Web Conference, December 1995

[LTP1986] LaLonde Wilf R., Thomas Dave A., Pugh John R.; An exemplar based Smalltalk; SIGPLAN Notices, 21, 11; OOPLSA'96, Prtland, OR, USA, Oct 1986

[LuA1990] Lucco Steven E., Anderson David P.; Tarmac: A Language system substrate based on mobile memory; DCS'90, Paris, France, June 1990; Pages 46-51

[LuH1998] Luff Paul, Heath Christian; Mobility in Collaboration; CSCW'98, Seattle, WA, USA, 14.11.1998; Pages 305-314w; ACM

[Lux1995a] Wolfgang Lux; Adaptierbare Objektmigration und eine Realisierung im Betriebssystem Birlix; GMD-Bericht, 252; Oldenbourg; 1995

[Lux1995b] Lux Wolfgang; Adaptable object migration: concept and implementation; SIGOPS, 29, 2, 4. 1995; Pages 54-69; ACM

[LYI1995] Lea Rodger, Yokote Yasuhiko, Itoh Jun-ichiro; Adaptive Operating System Design Using Reflection; LNCS, 1107; OBPDC'95, Tokyo, Japan, June 1995; Pages 205-218

[Mac1997] Machiraju Vijay; A Framework for Migrating Objects in Distributed Graphics Applications; Master Thesis; University of Utah; 1997

[MaD1999] Klein Mark, Dellarocas Chrysanthos; Exception Handling in Agent Systems; Autonomous Agents, Seattle, WA, USA, 1.5.1999; Pages 62-68; ACM

[Maf1993a] Maffeis Silvano; A Flexible System Design to Support Object-Groups and Object-Oriented Distributed Programming; Lecture Notes in Computer Science, 791; ECOOP'93, Kaiserslautern, Germany, July 1993

[Maf1993b] Maffeis Silvano; Electra - Making Distributed Programs Object-Oriented; Technical Report of the Institut for Informatics, IFI-93-17; Universisty of Zurich; 1993

[Mar1994] Martin Paul; The Formal Specification in Z of Task Migration on the Testbed Multicomputer; Edinburgh University; 1994

[MaS1988] Maguire G., Smith J.; Proess Migration: Effects on Scientific Computation; SIGPLAN Notices, 23, 3, Mar 1988; Pages 102-106

[MB+1998] Milojicic Dejan, Breugst Markus, Busse INgo, Campbell John, Covaci Stefan, Friedman Barry, Kosaka Kazuya, Lange Danny, Ono Kouichi, Oshima Mitsuru, Tham Cynthia, Virdhagriswaran Sankar, Shite Jim; MASIF - The OMG Mobiel Agent System Interoperability Facility; LNCS, 1477; MA'98, Stuttgart, Germany, September 1998; Pages 50-67

[McC1987] McCullough Paul L.; Transparent Forwarding: first steps; SIGPLAN Notices, 22, 12; 22, 12; OOPSLA'87, Orlando, FL, USA, Oct 1987; Pages 331-341

[MD+1998] Migliardi Mauro, Dongarra Jack, Geist Al, Sunderam Vaidy; Dynamic Reconfiguration and Virtual Machine Management in the Harness Metacomputing System; ISCOPE'98, Sanat Fe, TX, USA, December 1998

[Mey1992] Meyer Bertrand; Eiffel - The Language; Prentice Hall, New York, NY, USA; 1992

[Mil1987] Miller B.; DEMOS/MP: The Development of a distributed Operating System; Software Practice and Experience, 17, 4, April 1987; Pages 277-290

[Mil1994] Milojicic D.; Load Distribution; Vieweg, Braunschweig, Germany; 1994

[MKR1995] Mascarenhas E., Knop F., Rego V.; ParaSol: A Multi-threaded Syytem for Parallel Simulation Based on Mobile Threads; Winter Simulation Conference, Dec 1995; Pages 690-697

[MKR1996] Mascarenhas E., Knop F., Rego V.; Ariadne: Architecture of a portable Thread System Supporting Thread Migration; Software Practice and Experience, 26, 3, Mar 1996; Pages 327-357

[MMK1998] Matsubara Katsuya, Maekawa Takahiro, Kato Kazuhiko; Worldwide Component Scripting with the PLANET Mobile Object System; LNCS, 1368; WWCA'98, Tsukuba, Japan, March 1998; Pages 56-71

[MMN1993] MADSEN OLE LEHRMANN , MOLLER-PEDERSEN BIRGER , NYGAARD KRISTEN; Object-Oriented Programming in the BETA Programming Language; Addison-Wesley, Wokingham, England; 1993

[MMW1994] Mendelzon Alberto O. , Milo Tova , Waller Emmanuel; Object Migration; SIGMOD; Principles of Database Systems, Minneapolis, 1.5.1994; Pages 232-242; ACM, USA

[MNC1991] Masini Gerald, Napoli Amedeo, Colnet Dominique, Leonard Daniel, Tombre Karl; Object Oriented Languages; Academic Press, London, UK; 1991

[Moe1993] Mössenböck Hanspeter; Objektorientierte Programmierung in Oberon-2; Springer, Berlin, Germany; 1993

[MoV1993] Moons H., Verbaeten P.; Object Migration in a Heterogeneous World - A Multi-Dimensional Affair; Third International Workshop on Object Orientation in Operating Systems, North Carolina, Dezember 1993; Pages 62-72

[MVH1990] Moons H., Verbaeten P., Hollberg U.; Distributed computing in heterogeneous environments; EUUG'90, Munich, West Germany, 23-27 April 1990; Pages 15-25

[MWH1994] Moore Ivan, Wolczko Mario, Hopkins Trevor; Babel - A Translator from Smalltalk into Clos; Tools USA'94; Prentice Hall; 1994

[MZ+1993] Milojicic Dejan, Zint Wolfgang, Dangel Andreas, Giese Peter; Task Migration on the Top of the Mach Microkernel; Usenix Mach Symposium III, Santa Fe, April 1993; Pages 273-289

[NaD1992] Nascimento Claudio , Dollimore Jean; Behavior Maintenance of Migrating Objects in a Distributed Object-Oriented Environment; Journal of Object Oriented Computing, September 1992

[NaH1991] Nash C., Haebich W.; An Accidental Translator from Smalltalk to ANSI C; OOPS Messenger, 2, 3, 1991

[NC+1996] Nielson F., Cousot P., Dam M., Degano P., Jouvelot P., Mycroft A., Thomsen B.; Logical and Operational Methods in the Analysis of Programs and Systems; Lecture Notes in Computer Science; LOMAPS Workshop, Stockholm, Sweden, 24-26. 6. 1996; Pages 1-21; Springer, Berlin, Germany

[Nel1982] Nelson B. J.; Remote Procedure Call; CMU-81-119; Carnegie-Mellon-University; 1982

[Nel1987] Nelson Ted; Computer Lib / Dream Machines; Tempus Books; 1987

[NH+1999] Nestmann Uwe, Hüttel Hans, Kleist Josva, Merro Massimo; Aliasing Models for Object Migration; Lecture Notes in Computer Science, 1685; EuroPar'99, Toulouse, France, Auguts 1999; Pages 1352-1368

[Nie1987] Nierstrasz Oscar M.; Active Objects in Hybrid; OOPSLA'87, 1987; Pages 243-253

[NL+1998] Nisan Noam, London Shmulik, Regev Ori, Camiel Noam; Globally Distributed Computation over the Internet - The POCPORN Project; DCS'98; Pages 592-601; IEEE; 1998

[NN+1998] Nakamura Akihito, Nishioka Toshihiro, Hamazaki Yoichi, Tsukamoto Michiharu; Scalability in Object-Oriented Distributed Systems Environment OZ; LNCS, 1368; WWCA'98, Tsukuba, Japan, March 1998; Pages 72-87

[Nol1994] Nolte Jörg; Duale Objekte : Ein Modell zur objektorientierten Konstruktion von Programmfamilien für massiv parallele Systeme; GMD-Bericht, 232; Oldenbourg; 1994

[Nut1994a] Nuttall M.; A brief survey of systems providing process or object migration facilities; Operating Systems Review, 28, 4; Pages 64-79

[Nut1994b] Nuttall M.; Survey of systems providing process or object migration facilities; Imperial College Research Report DoC, 94, 10, November 1994

[NWM1993] Nicol J. R., Wilkes C. T. Manola F. A.; Object Orientation in Heterogeneous Distributed Computer Systems; Computer, 26, 6, June 1993; Pages 57-67

[OHE1996] Orfali Robert, Harkey Dan, Edwards Jeri; The Essential Distributed Objects Survival Guide; Jon Wiley, New York, USA; 1996

[OHK1987] O'Brian Patrick D., Halbert Daniel C., Kilian Michael F.; The Trellis programming environment; SIGPLAN Notices, 22, 12; OOPSLA'87, Orlando, Florida, USA, December 1987; Pages 91-102

[Ols1992] Olsen M. H.; A persistent object infrastructure for heterogeneous distributed systems; IWOOS'92, Dourdan, France, September 1992; Pages 49-56

[OpF1992] Opper Susanna, Fresko-Weiss Henry; Technology for Teams; van Nostrand Reinhold, New York, USA; 1992

[OtH1998] Othmann Mazliza, Hailes Stephen; Power Conservaton Strategy for Mobile Computers Using Load Sharing; Mobile Computing and Communications Review, 2, 1; Pages 44-51; 1998

[Pae1993] Paepcke Andreas; Object-Oriented programming - The CLOS Perspective; MIT Press, Cambridge MA, USA; 1993

[PaL1997] Park Anthony Sang-Bum , Leuker Stefan; A Multi-Agent Architecture Supporting Services Access; Mobile Agents, Berlin, Germany, 7.4.1997; Pages 62-73; Springer, Berlin, Germany

[Par1998] Parunak H. Van Dyke; What Can Agents Do in Industry and Why ?; LNAI / LNCS, 1435; CIA'98, Paris, France, July 1998; Pages 1-18

[PaS1975] Parnas D. L., Siwwiorek D. P.; Use of the concept of transparency in the design of hierarchically structured systems; CACM, 18, 7, July 1975; Pages 401-408

[PaS1991] Palsberg Jens, Schwartzbach Michael I.; Object-Oriented Type Inference; SIGPLAN Notices, 26, 11, 1991; OOPSLA'91

[PaS1994] Palsberg Jens, Schwartzbach Michael I.; Object-oriented Type Systems; Wiley, Chichester, UK; 1994

[PeG1999] Peter Yvan, Guyennet Herve; An Implementation of the Lifecycle Service Object Mobility on CORBA; PaCT'99, St. Petersburg, Russia, September 1999; Pages 282-295

[Pei1997] Peine H.; Ara -Agents for Remote Action; Prentice Hall, USA; 1997

[PeS1997] Peine Holger, Stolpmann Torsten; The Architecture of the Ara Platform for Mobile Agents; Mobile Agents, Berlin, Germany, 7.4.1997; Pages 50-61; Springer, Berlin, Germany

[PG+1996] Paoli D. De , Goscinski A., Hobbs M., Joyce P.; Performance Comparison of Process Migration with Remote Process Creation Mechanisms in RHODOS; Distributed Computing Systems, 1996; Pages 554-561

[PhZ1997] Philippsen Michael, Zenger Matthias; JavaParty - Transparent Remote Objects in Java; Concurrency: practice & Experience, 9, 11, November 1997; Pages 1225-1242

[Pic1998a] Picco G. P.; Understanding Code Mobility; Transactions on Software Engineering, 24, 5, May 1998; IEEE

[Pic1998b] Picco Gian Pietro; uCode: A Lightweight and Flexible Mobile Code Toolkit; LNCS, 1477; MA'98, Stuttgart, Germany, September 1998; Pages 160-171

[Ple1996] Pleier Christoph; Prozeßverlagerung in heterogenen Rechnerentzen basierend auf einer speziellen Übersetzungstechnik; Dissertationsschrift; Technische Universität, München, Germany; 1996

[PMC1992] NetClasses; PostModern Computing, Palo Alto, CA, USA; 1992

[PMR1999] Picco Gian Pietro, Murphy Amy L., Roman Gruia-Catalin; Lime: Linda meets Mobility; ICSE'99, Los Angeles, CA, USA, 16.-22. May 1999; IEEE

[PoM1983] Powell M. L., Miller B. P.; Process Migration in DEMOS/MP; Operating System Principles, Bretton Woods, New Hampshire, USA, October 1983; Pages 110-118; ACM

[PoW1985] Popek G. J., Walker B. J.; The LOCUS Distributed System Architecture; The MIT Press; 1985

[Rao1994] Rao Bindu R.; Object-Oriented Databases : Technology , Applications , and Products; McGraw-Hill, USA; 1994

[RaR1981] Rashid R. F., Roberstson G. G.; Accent: A communication oriented network operating system kernel; 8th Symposium on Operating System Principles, Pacifc Grooe, CA, USA, Dec 1981; Pages 64-75; ACM, New York

[RA+1998] Ranganathan M., Acharya Anurag, Andrey Laurent, Schaal Virginie; A Mobile Debugger for Mobile Programs; SIGMETRICS Symposium on Parallel and Distributed Tools, Welches, Oregon, USA, 3.8.1998; Pages 159

[RB+1998] van Roy Peter, Brand Per, Haridi Seif, Collet Raphael; A Lightweight Reliable Object Migration Protocol; Lecture Notes of Computer Science, 1686; ICCL'98, Chicago, IL, USA; Pages 32-46; 1998

[ReW1992] Reiser Martin, Wirth Niklaus; Programming in Oberon; Addison-Wesley, New York, NY, USA; 1992

[RH+1997] Roy Peter van, Haridi Seif, Brand gerd, Smolka Gert, Mehl Michael, Scheidhauer Rald; Mobile objects in Distributed Oz; Transactions on Programming Languges and Systems, 19, 5, Sep 1997; Pages 804-851

[RiH1998] Riely James, Hennessy Matthew; A Typed Language for Distributed Mobile Processes; POPL'98; Pages 378-390; 1998

[RiS1991] Richardson Joel, Schwarz Peter; Aspects: Extending object to support multiple independent roles; SIGMOD; Management of Data, May 1991

[RoC1996] Roush Ellard T. , Campbell Roy H.; Fast Dynamic Process Migration; ICDCS'96; Pages 637-645; IEEE, USA; 1996

[Röd1998] Röder Christian; Load Management Techniques in Distributed Heterogeneous Systems; Dissertation; Technische Universität, Munich, Germany; 1998

[Rou1995] Roush E.; The Freeze Free Algorithm for Process Migration; PhD Thesis; Universitsy of Illinois at Urbana-Champaign; 1995

[RT+1991] Raj Rajendra K., Tempero Ewan, Levy Henry M., Black Andrew P., Hutchinson Norman C., Jul Eric; Emerald: a general-purpose programming language; Software Practice and Experience, 21, 1, 1991; Pages 91-117

[SaM1998] Sahai Akhil, Morin Christine; Enabling a Mobile Network Manager through Mobile Agents; LNCS, 1477; MA'98, Stuttgart, Germany, September 1998; Pages 249-260

[Sat1998] Satyanarayanan M.; Mobile Computing: Where's the Tofu ?; Mobile Computing and Communication Review, 1, 1; Pages 17-21; 1998

[SBH1996] Straßer Markus, Baumann Joachim, Hohl Fritz; Mole - A Java Based Mobile Agent System; ECOOP'96 Workshop Object Systems, Linz, Austria, 8-9 July 1996; Pages 327-334; dpunkt, Heidelberg, Germany

[SB+1983] Stefik M., Bobrow D. G., Mittal S., Conway L.; Knowledge Programming in LOOPS: Report of an Experimental Course; The AI Magazine; Pages 3-13; 1983

[ScB1988] Schelvis Marcel, Bledoeg Eddy; The Implementation of a Distributed Smalltalk; LNCS, 322; ECOOP'86; Pages 212-232; 1988

[ScB2000] Schlichter Johann H., Borghoff Uwe M.; Computer Supported Cooperative Work; Springer, Germany; 2000

[Sch1990] Schill Alexander B.; Migrationssteuerung und Konfigurationsverwaltung für verteilte objektorientierte Anwendungen; Springer; 1990

[Sch1992d] Schill Alexander B.; Controlling Dynamic Object Transfer Between Distributed Repositories; PCCC'92, Scotsdale, AZ, USA, 1.4.1992; IEEE

[Sch1995] Schrimpf Harald; Migration of processes, Files and Virtual Devices in the MDX Operating System; SIGOPS, 29, 2, Apr 1995; Pages 70-81; ACM

[Sch1996] Schmidt Henning; Dynamisch veränderbare Betriebssystemstrukturen; GMD Bericht, 259; Oldenbourg; 1996

[ScM1993a] Schill Alexander B., Mock M. U.; DC++: Distributed Object-Oriented System Support on Top of OSF DCE; Distributed Systems Engineering Journal, 1, 2, 1993

[ScM1993b] Schill Alexander B., Mock Markus U.; Design and Implementation of Distributed C++; Springer aktuell; EURO-Arch'93, Munich Germany, 1993; Pages 469-482; Springer

[ScM1997] Schulze B., Madeira E.R.M.; Contracting and Moving Agents in Distributed Applications Based on a Service-Oriented Architecture; Mobile Agents, Berlin, Germany, 7.4.1997; Pages 74-85; Springer, Berlin, Germany

[SC+1986] Shaffert Craig, Cooper Topher, Bullis Bruce, Kilian Mike, Wilpolt Carrie; An Introduction to Trellis/Owl; SIGPLAN Notices, 21, 11, 1986; OOPSLA'86, Portland, Oregon, USA; Pages 9-16

[SC+1993] Sites Richard L., Chernoff Anton, Kirk Matthew B., Marks Maurice P. Robinson Scott G.; Binary Translation; CACM, 36, 2, Feb 1993; Pages 69-81

[SDP1991] Shrivastava S. K., Dixon G. N., Parrington G. D.; An Overview of the Arjuna Distributed Programming System; Software, January 1991; Pages 66-73; IEEE

[SGM1989] Shapiro Marc, Gautron Philippe, Mosseri Laurence; Persistence and Migration for C++ Objects; ECOOP'89, Nottingham, 10-14 July 1989

[SG+1989] Shapiro Marc, Gourhant Z., Habert Sabine, Mosseri L., Ruffin M., Valot C.; SOS: an object-oriented operating system - assessment and perspectives; Computing Systems, 2, 7, December 1989

[Sha1986b] Shapiro Marc; Structure and encapsulation in distributed systems: the Proxy Principle; Distributed Computing Systems, Cambridge, MA, USA, May 1986; Pages 198-204

[Shu1990] Shub Charles; Native Code process-originated migration in a heterogeneous environment; Computer Science Conference 1990, Washington DC, USA, February 1990; Pages 266-270; ACM

[SH+1998] Schill Alexander, Held Albert, Böhmka Wito, Springer Thomas, Ziegert Thomas; An Agent Based Application for Personalized Vehicular Traffic Management; LNCS, 1477; MA'98, Stuttgart, Germany, September 1998; Pages 99-111

[SiA1996] Silva Miguel Mira da, Atkinson Malcom; Combining Mobile Agents with Persistent Systems: Opportunities and Challenges; ECOOP'96 Workshop Reader, Linz, Austria, 7. 1996; Pages 335-339; dpunkt, Heidelberg, Germany

[Sie1996] Siegel Jon; CORBA Fundamentals and Programming; Jon Wiley, New York, USA; 1996

[Sil1996] Silva Miguel Mira da; Mobility and Persistence; Lecture Notes in Computer Science (LNCS), 1222; Mobile Object Systems, Linz, Austria, 8.7.1996; Pages 156-175; Springer, Berlin, Germany

[SiS1997] Silva Miguel Mira da, Silva A. Rodrigues da; Insisting on Persistent Mobile Agent Systems; Mobile Agents, Berlin, Germany, 7.4.1997; Pages 174-185; Springer, Berlin, Germany

[SmH1996] Smith Peter, Hutchinson Norman C.; Heterogeneous Process Migration: The Tui System; University of British Columbia, Vancouver, BC, Canada; 1996

[Smi1988] Smith J. M.; A Survey of Process Migration Mechanisms; Operating Systems Review, 22, 3, 1988; Pages 28-40

[Smo1995] Smolka Gerd; The Oz porgramming model; LNCS, 1000; Springer; 1995

[Sol1981] Sollins K. R.; Copying complex structures in a distributed system; LCS/TR-219; MIT, Cambridge, MA, USA; 1981

[SOM1994] Szyperski Clemens, Omohundro Stephen, Murer Stephen; Engineerig a Programming Language: the Type and Class System of Sather; LNCS, 782, 1994; Programming Language and System Architecture, Zurich, CH, March 1994; Springer

[SPM1992] Shalit Andrew, Piazza Jeffrey, Moon David; Dylan an object-oriented dynamic language; Apple Computer Eastern Research and Technology, Cambrigde MA, USA; 1992

[SPR1994] Sang J., Peters G., Rego V.; Thread Migration on Heterogeneous Systems via Compile-Time Transformations; ICPADS, 1994; Pages 634-639

[SSD1998] Silva Alberto, Silva Miguel Mira da, Delgado Jose; An Overview of AgentSpace; LNCS, 1477; MA'98, Stuttgart, Germany, September 1998; Pages 148-159

[Ste1990] Steele Guy L.; Common Lisp The Language (Second Edition); Digital Press, USA; 1990

[Ste1998c] Steindl Christoph; Intermodular Slicing of Object-Oriented Programs; LNCS, 1543; ECOOP'98 Workshop Code Analysis and Tools, Brussels, Belgium, July 1998; Pages 10-12

[Ste1998m] Steiner Maria Susanne; Lastverteilung in heterogenen Systemen; Universität Oldenburg

[StG1990a] Stamos J. W., Gifford D. K.; Remote Evaluation; Transactions on Programming Languages and Systems, 12, 4, October 1990; Pages 537-565

[StG1990b] Stamos J. W., Gifford D. K.; Implementing Remote Evaluation; Transactions of Software Engineering, 16, 7, July 1990; Pages 710-722; ACM

[StJ1995] Steensgaard Bjarne , Jul Eric; Object and native Code thread Mobility among Heterogeneous Computers; DIKU Rapport, 95, 20; 1995

[StO1996] Stoutamire David, Omohundro Stephen; The Sather 1.1 Specification; TR-96-012; ICSI, Berkeley, CA, USA; 1996

[Str1991] Stroustrup Bjarne; The C++ Programming Language; Addison-Wesley, Reading MA, USA; 1991

[Str1998] Stroustrup Bjarne; Die C++ Programmiersprache; Addison-Wesley, Bonn, Germany; 1998

[Su1991] Su Jianwen; Dynamic Constraints and object migration; VLDB'91, Barcelona, 1991; Pages 233-242

[Sun1992] The SELF Programmers's Reference Manual; Sun Microsystems; 1992

[Sun1999] Java 2 SDK version 1.2.2; Sun Microsystems; 1999

[SZM1994] Steketee Chris , Zhu Wei Ping , Moseley Philip; Implementation of Process Migration in Amoeba; DCS'94, Poznan, Poland, 21.6.1994; IEEE, USA

[TaV1996] Tardo J., Valente I.; Mobile Agents Security and Telescript; COMPCON'96; Pages 58-63; IEEE; 1996

[ThH1991] Theimer Marvin M., Hayes Barry; Heterogeneous Process Migration by Recompilation; DCS'91, Arlington, Texas, USA, 20-24.5.1991

[Thi1991] Thiel G.; Locus operating system, a transparent system; Computer Communications, 14, 6, 1991; Pages 336-346

[Tho1997] Thorn Tommy; Programming languages for mobile code; Computing Surveys, 29, 3, Sep 1997; Pages 213-239

[TH+1995] Tsukamoto Michiharu, Hamazaki Yoichi, Nishioka Toshihiro, Otokawa Hideyuki; The Version Management Architecture of an Object-Oriented Distributed Systems Environmnet: OZ++; OBPDC'95, Tokyo, Japan, June1995; Pages 310-328

[TK+1991a] Tanenbaum A. S., Kaashoek M. F., Renesse Robbert van, Bal H. E.; The Amoeba Distributed Operating System - A Status Report; Communcations of the ACM, 14, 6, July/August 1991; Pages 324-335

[TK+1991b] Tanenbaum Andrew S., Kaashoek M. Frans, Renesse Robbert van, Bal Henri E.; The Amoeba Distributed Operating System - A Status Report; Computer Communications, 14, 6, July 1991; Pages 324-335

[TK+1995] Tahara Yasuyuki, Kumeno Fumihiro, Ohsuga Akihiko, Honiden Shinichi; Formal Specification of Agent Evolution in Language Flage; OBPDC'95, Tokyo, Japan, June1995; Pages 329-348

[TK+1999] Tripathi Anand R., Karnik Neeran M, Vora Manish, Ahmed Tanvir, Singh Ram; Mobile Agent Programming in Ajanta; DCS'99, May 1999

[TLC1985] Theimer M., Lantz K., Cheriton D. R.; Preemptable Remote Execution Facilities for the V-System; Operating System Principles, 1th ACM Symposium on, Orcas Island, WA, USA, December 1985; Pages 2-12; ACM

[ToP1994] Tok Wang Ling, Pit Koon Teo; From Object Migration to Message Processing: a Re-Lock at Object Identity; Advances in Computing Techniques, Singapore, 1994

[ToP1995] Tok Wang Ling, Pit Koon Teo; Object Migration in ISA Hierarchies; DASFAA'95, Singapore, 10.4.1995; Pages 292-299

[TrK2000] Tripathi Anand R., Karnik Neeran M.; Mechanisms for Delegation of Privileges to Mobile Agents in Ajanta; ECOOP'2000 Workshop Mobile Object Systems, Sophia Antipolis, France, June 2000

[Tsc1996b] Tschudin Christian F.; The Messenger Environment MO - A Condensed Description; Lecture Notes in Computer Science (LNCS), 1222; Mobile Object Systems, Linz, Austria, 8.7.1996; Pages 149-156; Springer, Berlin, Germany

[Tuo1999] Tuosto Emilio; An Ada95 Implementation of an Network Coordination Language with Code Mobility; Lecture Notes in Computer Science, 1622; Ada Europe '99, Santander, Spain, June 1999

[UnS1987] Ungar David, Smith R. B.; Self: the power of simplicity; SIGPLAN Notices, 22, 12, Dec 1987; OOPSLA'87; Pages 227-242

[Veg1986] Vegdahl S. R.; Moving structures between Smalltalk images; SIGPLAN Notices, 21, 11; OOPSLA'86; Pages 466-471; 1986

[ViC1998] Vitek Jan, Castagna Guiseppe; Seal: A Framework for Secure Mobile Computations; LNCS, 1686; ICCL'98 , Chicago, IL, USA, May 1998; Pages 47-77

[Vit1996] Vitek Jan; Secure Object Spaces; ECOOP'96 Workshop Reader, Linz, Austria, 7. 1996; Pages 340-347; dpunkt, Heidelberg, Germany

[Voj1993] Vojik Franz; Autonome Veränderung des Replikationsgrades in verteilten Dateisystemen; Dissertation; Technische Universität München, Institut for Informatik, München, FRG; 1993

[VST1997] Vitek Jan, Serrano Manuel, Thanos Dimitri; Security and Communications in Mobile Object Systems; Lecture Notes in Computer Science (LNCS); Mobile Object Systems (MOS'96), Linz, Austria, July 1996

[WAA1998] Wu Daniel, Agrawal Divyakant, Abbadi Amr El; StratOSphere: Mobile Processing of Distributed Objects in Java; MobiCom'98, Dallas, TX, USA; Pages 121-132; ACM; 1998

[WB+1998] Wicke Christian, Bic Lubomir F., Dillencourt Michael B., Fukuda Munehiro; Automatic State Capture of Self-Migrating Computations in Messengers; LNCS, 1477; MA'98, Stuttgart, Germany, September 1998; Pages 68-79

[WC+1974] Wulf W., Cohen E., Corwin W., Jones A., Levin R., Person C., Pollack E.; Hydra: The Kernel of a Multiprocessor Operating System; Communications of the ACM (CACM), 17, 6, June 1974; Pages 347-345

[WDH1989] Weiser Mark, Demers Alan, Hauser Carl; The portable common runtime approach to interoperability; Operating System Principles (Twelfth Symposium), 1989; Pages 114-122

[Weg1990] Peter Wegner; Concepts and Paradigms of Object-oriented Programming; OOP Messenger, 1, 1, August 1990

[Wei1999] Weiss Gerhard (Editor); Multiagent Systems; MIT Press, Cambridge, MA, USA; 1999

[Whi1997] White James E.; Mobile agents; in: Software Agents (Bradshaw Jeffrey M.); Pages 437-472; MIT Press; 1997

[WoJ1995] Wooldrige M., Jennings N. R.; Intelligent Agents: Theories, Architectures and Languages; Lecture Notes in Articifical Intelligence, 890; Springer; 1995

[WPM1999] Wong D., Paciorek N., Moore D.; Java-based mobile agents; CACM, 42, 3, Mar 1999; Pages 92-102; ACM

[WP+1983] Walker B. J., Popek Gerald, English Robert, Kline Charles, Thiel Greg; The LOCUS Distributed Operating System; Operating System Principles, 17, 5, December 1993; Pages 49-70; ACM

[WP+1997] Wong David, Paciorek Noemi, WalshTom, DiCelie Joe, Young Mike, Peet Bill; Concordia: An Infrastructure for Collaborating Mobile Agents; Mobile Agents, Berlin, Germany, 7.4.1997; Pages 86-97; Springer, Berlin, Germany

[WZF1994] Wortmann D., Zhou S., Fink S.; Automating data Conversion for Heterogeneous Distributed Shared Memory; Software Practise and Experience, 24, 1, Jan 1994; Pages 111-125

[YS+1988] Yankelovich Nicole, Smith Karen E., Garrett Nacy, Meyrowitz Norman; Issues in Designing Hypermedia Document Systems; Microsoft Press; 1988

[Zay1987a] Zayas E.; Attacking the Process Migration Bottleneck; Operating System Principles - 11th Symposium on, November 1987; Pages 13-24; ACM-SIGOPS

[Zho1992] Zhou; Utopia - A Load Sharing Facility for large heterogeneous distributed Computer Systems; TR CSRI-257; University of toronto; 1992

[ZS+1992] Zhou S., Stumm S., Li K., Wortmann D.; Heterogeneous distributed shared memory; Transactions on Parallel and Distributed Systems, 3, 5, Sep 1992; Pages 540-554