Synthesis of distributed systems from synchronous dataflow programs

Jan Romberg

Institut für Informatik
der Technischen Universität München

Synthesis of distributed systems from synchronous dataflow programs

Jan Romberg

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

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Alois Knoll
Prüfer der Dissertation:	
1.	Univ.-Prof. Dr. Dr. h. c. Manfred Broy
2.	Univ.-Prof. Dr. Klaus D. Müller-Glaser Universität Karlsruhe (TH)

Die Dissertation wurde am 15.03.2006 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 27.06.2006 angenommen.

Abstract

Synchronous dataflow languages are a popular tool for systems specification in domains such as real-time control and hardware design. The potential benefits are promising: Discrete-time semantics and deterministic concurrency reduce the state-space of parallel designs, and the engineer's intuition of uniformly progressing physical time is clearly reflected. However, for deriving implementations, use of synchronous programs is currently limited to hardware synthesis, generation of non-distributed software, or deployment on time-triggered architectures. For distributed software systems based on event-triggered bus systems and on-line schedulers, it is still an open problem how conformance with an abstract synchronous design is to be defined.

This thesis examines both the problem of synthesis, and the problem of behavioral preservation, for distributed implementations of synchronous programs. A simple synchronous language is presented: this class of languages is shown to meet some essential application requirements, such as boundedness of computational resources, or compositionality. For programs with appropriate delays at subsystem boundaries, two complementary implementation schemes suited for the OSEK operating system and event-triggered communication media, respectively, are presented. Both implementation schemes are characterized by the fact that individual steps of the synchronous program's semantics are spread out in time, or linearized.

For such linearized implementations, we provide a general formal framework, where the synchronous program and its linearized implementation are captured both on the level of language (words) and on the level of transition structures (automata). As an example application of the theory, it is shown how certain behavioral properties expressed in the temporal logic LTL can be preserved from the synchronous program to its implementation. A simple method is provided for checking whether a given property is preservable.

Kurzfassung

Synchrone Datenflusssprachen sind ein etabliertes Mittel der Systembeschreibung in Anwendungsdomänen wie Regelungs- und Steuerungssystemen oder Hardwareentwurf. Diese Klasse von Sprachen bietet einige bekannte Vorteile, wie zum Beispiel einen gegenüber anderen Ansätzen reduzierten Zustandsraum bei der Beschreibung paralleler Systeme. Für die Synthese von Implementierungen ist die Verwendung von synchronen Datenflusssprachen momentan vor allem auf den nichtverteilten Fall oder auf Implementierungsplattformen mit starken Synchronisationsannahmen eingeschränkt. Dagegen existieren für schwächer synchronisierte Plattformen nur wenige Arbeiten zur semantikerhaltenden Implementierung eines synchronen Programmes.

Die vorliegende Arbeit untersucht das Problem der verteilten Implementierung synchroner Programme. Zu diesem Zweck wird eine einfache synchrone Datenflusssprache definiert: es wird gezeigt, wie diese Sprachen einige typischen Anforderungen der Anwendungsdomäne erfüllt. Für synchrone Datenflussprogramme werden zwei komplementäre Implementierungsstrategien aufgezeigt, wobei die eine auf den automobilen Betriebssystemstandard OSEK basiert, während die andere für ereignisgesteuerte Kommunikationsmedien geeignet ist. Beide Mechanismen sind dadurch gekennzeichnet, dass die einzelnen Berechnungen in kausal geordneten Einzelschritten, oder linearisiert, erfolgen.

Für solche linearisierten Implementierungen wird in der Arbeit eine allgemeine Theorie definiert, die ein synchrones Programm und seine Linearisierung sowohl auf der Ebene von Abläufen (Wörtern) als auch auf der Ebene konkreter Transitionssysteme (Automaten) abbildet. Als beispielhafte Verwendung der Theorie wird mit Hilfe der Temporallogik LTL demonstriert, wie bestimmte Verhaltenseigenschaften eines synchronen Programms für die Linearisierung des Programms erhalten bleiben. Dazu wird eine einfache Methode angegeben, mit der die Erhaltbarkeit einer Eigenschaft überprüft werden kann.

Danksagung

Mein Dank für die Unterstützung im Rahmen dieser Arbeit geht zunächst an meinen Doktorvater, Prof. Manfred Broy, der hier an der TU München eine einzigartige und sehr freie Umgebung für wissenschaftliches Arbeiten geschaffen hat, und mir mit Rat und Tat zur Seite stand. Prof. Klaus D. Müller-Glaser gebührt der Dank für die Übernahme des Zweitgutachtens.

Meine Frau Theresia und meine Söhne Jakob, Paul und Benedikt haben mich in jeder Phase dieser Arbeit sehr liebevoll unterstützt, und mussten dafür stellenweise meine körperliche An- sowie geistige Abwesenheit daheim ertragen. Vielen Dank dafür. Besonderer Dank geht an Benedikt, der mich in unseren vielen zweisamen Stunden doch meistens gewähren ließ (mit Ausnahme einer kaputten Taste am Rechner, die durch seine begeisterte Mitwirkung daran glauben musste.) Mein Bruder Tim ist für mich schon immer eine Quelle geistiger Inspiration, und hat den Entstehungsprozess dieser Arbeit zumeist telefonisch mitverfolgt.

Mit meinen Kollegen Andreas Bauer, Peter Braun, Martin Leucker und Bernhard Schätz habe ich einzelne Punkte dieser Arbeit besprochen, und habe durch ihre Einwirkung immer mal wieder einen neuen Gedanken aufgegriffen oder verworfen. Als "Startvektor" für meine Zeit am Lehrstuhl habe ich durch den Kontakt zu Alexander Pretschner und Chris Salzmann sehr profitiert. Unseren Projektpartnern im AutoMoDe-Projekt, Ulrich Freund, Pierre Mai und Dirk Ziegenbein, gilt mein Dank für viele interessante Diskussionen und Anregungen.

Andreas Bauer, Christian Kühnel und Stefan Wagner haben Teile dieser Arbeit gelesen und viele gute Anmerkungen gegeben. Danke euch dafür.

Contents

Chapter 1

Introduction

Reasoning about the correctness of programs, and exploring adequate ways to represent them, has been at the heart of activities in computer science ever since its inception. In the theoretical field, the last decades have seen a shift of focus from research about sequential programs to formalization and verification of concurrent, distributed applications. To grasp the behavior of a distributed application poses a particular challenge to the human mind, as these systems are capable of a large variety of possible internal choices, or non-determinism. The consequence is that distributed systems are considered hard to develop, verify, and maintain.

At the same time in industry, the embedded control systems sector has transformed dramatically. Firstly, the overall *amount* and *complexity* of electronics and software is steadily increasing, culminating in a software size of 10 million lines of code (LOC) in a present-day luxury automobile. For illustration, this is roughly one third the size of a modern-day PC operating system, with notably higher reliability requirements, and within a much more heterogeneous technical setting. Secondly, the *connectivity* of formerly isolated functions has increased at a similar pace, such as integration of engine management, chassis control, and interior body electronics to integrated functionalities in automobiles, or combination of formerly independent cockpit avionics and flight control units to sophisticated fly-by-wire controls in aviation.

Obviously, increased complexity and connectivity pose new verification and integration challenges. Understanding, debugging, and servicing complex and connected systems is hard: for instance, an estimated 70% of all electronic failures occuring in vehicles are of sporadic nature [Ele05], meaning they cannot be easily reproduced. To illustrate the commercial magnitude of the problem, the percentage of electronics-related causes in vehicle breakdowns has continually increased to nearly 60% in the last

years [Dud04], and 17% of all car owners interviewed in a study had experienced problems with electronic controls in their vehicles [Kfz04]. Given the proportion of the current problem in embedded control systems, and given the ambition of computer science to represent and understand distributed functionality, it is therefore quite natural to ask how the theoretical work may influence the developments in practice.

As a contribution, this thesis relates a particular programming model for distributed systems, *synchronous dataflow programs*, to the particular architectures and constraints of an industrial domain, *automotive control systems*, and related domains.

1.1 Problem Statement

The domain: Automotive control systems. Embedded control systems continuously respond to incoming physical and user input by interfering with the environment e. g. through actuators. Typical examples for automotive control systems include drivetrain (engine, gearbox) management, chassis controls, or interior body controls. The following *application* aspects of automotive control systems are centrally considered in this thesis:

Resource-bounded algorithms. Applications in automotive control typically do not rely on features that cannot be tightly bounded in terms of timing and memory consumption. Such dynamic features would include recursive function calls and datatypes, or dynamic memory and process creation. Control algorithms can typically be written without dynamic features: iteration/recursion is restricted to traversals of fixed-size data structures, and dynamic memory/process creation is not needed.

Time is essential and uniform. Time is considered an essential part of modeling the application. This is true, for instance, if an integration algorithm with reference to physical time is used, or if a watchdog timer is employed, which may elapse after a given time interval. Control algorithms for physical plants are tightly dependent on the passage of time in the system's environment, and are therefore often designed with respect to some uniform notion of time across the system.

Throughout this thesis, the term *platform* comprises technical infrastructure like processors, networks, operating systems, sensors, and actuators. On the level of platforms, the following aspects are seen as central to the application domain:

High efficiency demands. Large production volumes and the focus on per-unit costs necessitate a minimum of computational resources with maximum utilization. This need for efficiency calls for highly optimized implementation techniques for applications, and is also clearly reflected in typical automotive platform technologies. As a concrete technical example, the automotive operating system ERCOS^{EK}[PMSB96] uses a specific wait-free scheme for inter-process communication, which sacrifices inter-task synchronization for speed, minimizing inefficiencies due to context changes and blocking times. Secondly, the upcoming AUTOSAR [AUT03] communication infrastructure uses statically generated communication layers, sacrificing some dynamic behavior features for performance.

Firmly established platforms. In automotive systems, for a number of reasons, there is a strong tendency to remain committed to existing platform technologies, such as bus systems or operating systems. Platforms cannot be switched on-the-fly to adapt to new paradigms of component definition and composition. This is a noteable difference to domains such as client devices in embedded telecommunications or web services, where a much higher fluctuation rate in platform technologies can be afforded. Consequently, our approach needs to be particularly suited for the established platforms in the automotive domain, such as OSEK [OSE01] and CAN [Ets01], and related technologies.

Need for simplicity. Virtually no one will deny the virtue of simple and effective languages and conceptual models in software engineering. For instance, much of the progress in software development productivity since the machine language age can be attributed to the availability of high-level languages and compilers. In this context, one may adopt Brooks' [Bro87] qualitative differentiation between "essential complexity" for the difficulties inherent in the nature of a given software, and "accidental complexity" for the noninherent characteristics having more to do with a particular way of producing the software. Clearly, though formal languages are not a panacea for the more universal problems of software engineering, they should on their part minimize the need for expressing accidental information, and should allow developers to concentrate on the essentials of an application. A good language can reduce the number of concepts necessary to express a given operation, data element, or relationship. In support of this argument, most studies on quantifying development effort [Alb79][Mac94] have accepted an intuitive association between (measurable, language-related) aspects of system size and interconnectivity on the

one hand, and effective product complexity and development effort on the other.

For illustration how accidental information may affect productivity, it shall be worthwile to look at an example:

> **Example (Simplicity vs. productivity).** Consider firstly, the difference between an integer modulo operation in a high-level language, and the corresponding sequence of register load, addition, comparison, subtraction, and register store operations in assembly language. The assembly code version is harder to get right on the first try, and may be written differently by different developers although the same task is performed, causing difficulties in understandability and maintenance. As a second example on a higher level, consider the case of control systems engineering: block diagram languages such as Simulink [MW] capture domain-specific design paradigms in a simple-to-use, visual fashion, and provide reusable, pre-validated library blocks for standard computations that are effectively part of the language itself. A discrete transfer function, for instance, is one library block with a number of configurable parameters in Simulink, whereas a C implementation of the discrete transfer function may involve several tens of lines of code, again with a large accidental variety in possible solutions.

Need for formal foundation. For a given (suitable) language, the need for a formal foundation of the language is evident. Formal foundation means that both allowable language concepts and their interrelations (*syntax*), as well as their meaning, for instance in terms of behavior or timing (*semantics*), are well-defined and clear. Formal need not necessarily mean mathematical, but it is largely undisputed that the established mathematics-oriented ways of capturing syntax and semantics have contributed greatly to the understanding and development of languages and their foundations.

In the field of sequential languages such as C/C++, Ada, or Java, most syntax and semantics features are intuitively well-understood by trained developers. Sequential languages rely on the von Neumann machine model, where a program modifies a shared memory area through successive statement. The von Neumann model, along with language features such as static typing (e. g. Java) or procedural and object-oriented programming, is strongly represented in engineering and computer science education. In stark contrast to this situation, the field of distributed embedded systems lacks a commonly agreed “language” for system composition at

a level higher than individual statements, threads, and low-level synchronization/communication primitives. Therefore, as of today, building distributed embedded systems is radically different from building sequential programs: both syntax and semantics are not explicitly defined in many application contexts, and therefore unclear.

Without a language for system construction, the allowable syntax for high-level composition are undefined, or only very implicitly captured by a number of heterogeneous information sources. Similarly, the semantics of system interfaces and composition is unclear in such a setting. The lack of a high-level language for system composition forces detailed knowledge of the code, of configuration parameters of the platform such as thread and message priorities or frequencies, and of causal relationships, in order to obtain a rudimentary understanding of communication and activation semantics on a higher level.

Example (Lack of language illustrated). As an illustrating example for lack of syntax, two engineers designing software components A and B, respectively, may decide to implement a FIFO communication link between A and B. Without an appropriate language, this chosen high-level "language construct" will be buried in the code of A and B, along with possibly some database entries, textual documents, or configuration files. The precise semantics of the above communication link, such as timing and reliability of message transport, and encoding of message contents, cannot be easily defined on a higher level if no appropriate language exists. As a consequence, a thread for component B receiving a message from a thread for component A, running on some other processor, may or may not reliably receive the message after emission.

To improve on this situation, standardization efforts such as AADL [Soc04], AUTOSAR [AUT03], or SysML [Sys05], are underway, giving syntax-oriented definitions of domain-specific languages for high-level composition. An appropriate high-level *semantics* for communication and synchronization in distributed embedded control systems is so far not an integral part of these efforts: besides the work in the AADL context on hybrid automata formalization[Ves00], the standardization efforts do not yet provide a tightly defined computational model, or ignore semantic issues completely.

Need for compositionality. In mathematics and semantics, the principle of compositionality states that the meaning of a complex expression is determined by a combination of (1) the meaning of its constituent expression

and (2) some given rules to combine the individual meanings. While there are well-known examples for non-compositional theories on a purely theoretical level, when applying this idea to the incremental construction of a software system, the theoretical compositionality criterion can often be fulfilled in some way, but the nature of the rules is important. Composition rules for a scaleable approach to software construction should correspond to some intuitive expectation of the developer, and interference between components should be restricted accordingly. An effective compositional methodology should yield the ability to build and understand large systems by repeating the task of building (or plainly using) and understanding smaller ones. We shall illustrate in Chapter 2 how compositionality can be related to aspects of languages for software composition.

Need for disciplined refinement. In the process of gradually designing a complex embedded system, it has been recognized that there is a strong need for varying modeling paradigms and degrees of precision at different development phases [Bro93][BRS00][AAR05]. The spectrum ranges from informal, weakly structured, largely textual requirements models to detailed behavioral models of a system. This thesis shall be concerned with the "lower" end of this spectrum: detailed, behaviorally defined models of embedded control systems, which may serve as a basis for verification and validation. Such behaviorally defined models or programs are, in turn, abstractions of their corresponding implementations. The usefulness of such abstractions in the development process strongly hinges to the ability to relate models and their refinements in a *disciplined* fashion [Sel03]. As a consequence of the growing system complexity, *behavioral* properties, such as a verdict on semantic interoperability of two software components, are especially difficult to relate and verify. By relating abstract and refined behaviors in a well-defined and scaleable manner, a refinement discipline is a central enabler for early validation and verification based on system models.

Structure and behavior of abstract models and their refined implementations can related in different ways. The methodical background of this work is mainly *correct-by-construction* or *top-down synthesis* of implementations from abstract programs, or models. In a correct-by-construction approach, an implementation is obtained from an integrated model of the system by a synthesis procedure. The procedure is sufficiently automated and/or formalized so that a high degree of confidence in the preservation of essential properties along the synthesis process is justified.

The methodical dual of top-down synthesis are various forms of *con-*

formance checking, or *bottom-up* verification [BJK+05][RWNH98][dBOP+00] [RH05]. Bottom-up methods compare an abstract program with an (often independently constructed) implementation with respect to a given, normative refinement relation. Comparison may be complete, such as in formal verification, or incomplete, such as in monitoring or testing. Comparison may also be offline, such as in testing or formal verification, or online, such as in monitoring. Again, the preservation of essential properties is the core concern, and the choice of both refinement relation and considered scenarios in incomplete verification depend on the properties of interest. The top-down and bottom-up approaches can be seen as complementary, each with its specific advantages and disadvantages.

1.2 Contributions

Review of synchronous dataflow for software construction. This thesis gives a overview of the synchronous dataflow model, and examines the dataflow paradigm with respect to some application-related concerns such as simplicity, suitability for architecting, and suitability for programming. The class of systems considered in this thesis, distributed, software-intensive systems, is somewhat different from the more traditional, accepted domains of synchronous programming, such as hardware design, and (local) control algorithm design [BCE+03].

Correct-by-construction methods. A correct-by-construction method for synchronous dataflow models yields correctness obligation on different granularity levels of a distributed software system. In automotive applications, software is partitioned into tasks, several of which may be running on a single processor. Several processors, in term, are combined over a communication medium, such as a bus system. Correctness obligations with respect to communication and synchronization appear on the intra-task level, inter-task level, and inter-processor level. Ensuring correct intra-task communication and synchronization as part of sequential code synthesis for synchronous dataflow models is state of the art [HRR91][ETA][MW][dSP]. For each of the remaining two levels of inter-task and inter-processor communication and synchronization, the thesis provides suitable methods for correct-by-construction implementation of synchronous dataflow models.

The first method, outlined in Section 3.2, considers the inter-task level on a single processor. Semantics-preserving inter-task communication based on preemptively scheduled tasks with fixed priorities is achieved us-

ing wait-free inter-process communication (IPC). In wait-free IPC, the operating system uses a double- or triple-buffering technique to ensure that a reader task will have consistent input values for the duration of its activation. While there are a number of inter-task communication primitives known from the literature which ensure atomic and consistent accesses to shared data, wait-free IPC has emerged as the mechanism of choice for control law computations in resource-restricted applications [PMSB96]. The singleprocessor implementation scheme is based on additional assertions with respect to presence/absence patterns of message streams (clocks). Using our method, one can implement a zero-delay or unit-delay communication link specified in the synchronous dataflow model. The kind of implementable delay depends on the relation of the sender and receiver clock.

As part of the AutoMoDe project [BBR+05], the singleprocessor method has been implemented in an automatic translator from the synchronous AutoFOCUS tool [HSE97] to ASCET [ETA] designs.

The second method, described in Section 3.3, is suited for the interprocessor level. Preservation of communication and synchronization in networks is facilitated by a common time base, and by using established bounds on message latencies. Event-triggered bus systems such as Controller Area Network (CAN), are not designed for providing a common, fault-tolerant time base. Furthermore, message latencies cannot be locally determined. The multiprocessor method presented therefore relies on a simple, fault tolerant synchronization layer, which uses application data traffic to keep nodes loosely synchronized, and a technique for estimation of message latencies based on global knowledge about periodicities and size of application messages [TB94]. Our method then guarantees semantics-preserving implementation under given operational conditions.

One novelty of both methods over other approaches for compositional real-time systems design, such as time-triggered architectures [Kop97], is the fact that a possibly wider set of platforms is supported. To achieve this, a method spanning both application development (with synchronous dataflow models) and platform is used, instead of a particular platform.

Refinement relations. A suitable definition of a refinement relation is at the heart of both top-down and bottom-up approaches. Clearly, such refinement relations are of some theoretical interest by themselves, independently of their actual methodical use. This thesis describes a particular refinement relation in Chapter 4, which reflects implementations of

synchronous dataflow models based on subprogram threads communicating through channels of finite size. For this purpose, the notion of linearization is introduced. We then formalize the refinement relation on two levels: firstly, between runs of synchronous programs and their linearizations by means of a map between synchronous words and their linearized counterparts. Secondly, by the definition of synchronous automata and linearized automata, and a construction rule for the latter, based on their synchronous equivalent. The two formalizations for languages and automata are shown to be coincident.

Property preservation. The refinement relation etablished in Chapter 4 is employed in the same chapter to demonstrate how some established property of a synchronous program in the temporal logic LTL implies an equivalent property for the linearization. This kind of implication is generally known as property preservation, and is an essential ingredient of any disciplined method for refinement. We sketch a simple algorithm that decides whether a given LTL formula is preservable according to our framework. The method utilizes knowledge about immediate dependencies between the synchronous program's variables, which is a byproduct of the mandatory causality analysis performed on synchronous programs.

1.3 Thesis Outline

Chapter 2 motivates and introduces a simple synchronous dataflow language, which is the formal basis for the refinement-related work in Chapters 3 and 4. The concrete choice of computational model is supported by a review of the particular features of dataflow languages, and by an evaluation of the synchronous dataflow model with respect to architecting and programming of software. Subsequently, the syntax and semantics of SSDL is defined, which is a very simple synchronous dataflow language, including the particular aspect of causality analysis.

Chapter 3 describes two implementation schemes for synchronous dataflow programs: firstly, we define a singleprocessor implementation scheme, where a synchronous dataflow program is partitioned into several threads running on one processor, and the composite multithreaded program implements the semantics of the synchrous dataflow program using interthread communication primitives. Secondly, an implementation scheme for the multiprocessor case is described, where a synchronous dataflow

program is split across several processors communicating over some event-triggered, bounded-jitter communication medium. Both implementation schemes are shown to preserve certain critical aspects of the dataflow program's semantics.

Chapter 4 defines a general formal refinement relation between a synchronous dataflow program and its implementation based on threads and finite channels. For this purpose, the notion of linearizations both on the level of languages (words) and operational models (automata) is introduced. Based on the refinement relation between words of the synchronous program and their linearization, it can be shown that a restricted class of behavioral properties in the temporal logic LTL is preserved from synchronous dataflow programs to their linearizations.

Chapter 5, finally, gives a conclusion and an outlook of this thesis for possible future work.

Chapter 2

Synchronous Dataflow Programs

This chapter motivates and introduces a synchronous dataflow language called SSDL, which will form the formal basis for the work on distribution in Chapters 3 and 4. In Section 2.1, we justify our choice of computational model by giving a short historic account of the development of dataflow languages and their particular features, and by evaluating the dataflow model, in particular the synchronous dataflow model, with respect to its suitability for various software engineering tasks. Section 2.2 defines the syntax and semantics of SSDL, which is a very simple synchronous dataflow language. The special aspect of causality analysis is discussed: this analysis is used to identify and reject invalid SSDL programs. Section 2.3 briefly summarizes some related publications.

2.1 Rationale

Before delving into the definition of a synchronous dataflow language, it will be fruitful to analyze how the dataflow paradigm came about, and to shed some light on the reasons why this computational paradigm may be more useful than others in the targeted domain of automotive control systems, or software engineering for real-time systems in general.

2.1.1 Dataflow: A brief history

Characteristics. The currently most popular usage of the term dataflow is in software engineering, where it generally refers to the flow of information between data processing entities, or components. Earlier uses of the term dataflow stem from parallel computing. Unlike controlflow-oriented computing models such as the popular sequential programming

languages, the dataflow paradigm focuses on the directed exchange of data between units of computation, or *processes*. A dataflow program can be visualized as a directed (multi)graph whose nodes denote computing processes, and whose edges denote data dependencies between processes. Among most prominent characteristics of dataflow languages are *freedom from side effects*, *flexibility with respect to scheduling*, and various forms of *totality*.

Freedom from side effects. The result of a process is solely dependent on its inputs, and there is no other (more or less implicit) form of communication. In the related area of functional languages, this property is often referred to as "referential transparency". In contrast, most sequential languages (C, Java) rely on the von Neumann model, where a global state visible in different regions of the control flow (and visible to several functions) makes both the dataflow and the order-dependence between different parts of a program implicit.

Flexibility with respect to scheduling. The possible orderings for executions of a program are directly defined by the data dependencies. Thanks to this property, for instance, one does not need explicit synchronization constructs in the language. Within the allowed orderings, the result of a computation does not dependent on the particular chosen order of evaluation: data values in a dataflow language cannot be updated by side effects. Dataflow languages admit *parallelism*: For instance, for evaluating $f(g(x), h(x))$ in an eager (data-driven) dataflow language, one can evaluate $g(x)$ and $h(x)$ simultaneously.

Totality. A dataflow program in (most) dataflow and functional languages for the embedded sector will provide a response to any input, thus constitutes a total function over its inputs. This corresponds to the intuitive notion of a *reactive* program. Totality is compromised if the language allows unbounded recursion, where termination is undecidable. Thus, only languages with restricted forms of recursion, such as the SSDL language defined in Section 2.2, satisfy the totality criterion.

Development. Historically, there have been several well-known dataflow languages [AW77, MS83]. Synchronous dataflow languages shortly followed, with the definition of LUSTRE and SIGNAL in the 1980s ([BCE+03] is a recent paper)[1]. For an up-to-date, somewhat programming-centric survey of dataflow languages, see [JHM04].

[1]The term "synchronous dataflow" is also used for the Lee/Messerschmitt model [LM87], which is well-known in the signal processing community. This model shares

Dataflow got its theoretical underpinnings when Kahn [Kah74] defined dataflow programs as sets of recursion equations, and used the least fixed point to characterize their behavior. Kahn's work is based on the denotational or Scott/Strachey approach [SS72], which generally uses fixed point theory on various forms of partial orders to characterize computations, and yields mathematically tractable and compositional semantics for a number of computational paradigms [Win93].

Many of the features of dataflow languages are actually shared with *functional* or *applicative languages*, which had been developed since the 1950s based on foundational work by Church in the 1930s. The tight relation between functional or applicative programming with dataflow programming has been noted by countless authors. It comes as no surprise that there is both a wealth of functional language researchers having pushed their approaches towards the use as a parallel language, [AW77, MS83, Nik90, CCL91] as well as dataflow languages being embedded in functional languages [Ree95] [CP96].

The precise difference between a "functional" and a "dataflow" language is somewhat blurry. As a tendency, most classical dataflow languages tend to be stronger on target-specific considerations, such as optimization for specialized target architectures. On a language level, the atomic unit of computation in dataflow is a process, the primary communication mechanism is through streams, and recursion may be restricted to special forms.

Functional languages, on the other hand, typically put more emphasis on type systems and advanced language features such as higher order functions, are less specialized with respect to communication, and allow more generic forms of recursion. The atomic unit of computation is the *operator*.

Programming and specification. Traditionally, in the 1970s and 1980s, the major goal in conceiving dataflow languages had been to provide a suitable programming model for programming dataflow hardware [JD75], a computing architecture that was quite popular in academia at the time. However, the quest to replace the predominant von Neumann architecture by dataflow hardware never materialized.

Clearly, the failure to establish dataflow on the hardware level does not necessarily diminish the conceptual quality of the dataflow programming

some characteristics with [BCE+03], such as static predictability, but is generally less expressive with respect to multirates, and is typically observed in an untimed fashion, similar to the linearizations described in Chapter 4.

model on the application or software level. In fact, as a consequence of the continuing dominance and superior efficiency of the von Neumann processor, dataflow researchers have shifted their attention to implementation of dataflow programs based on *multithreaded* architectures [PT91]. In a multithreaded implementation of a dataflow program, the parallelism inherent in the program is exploited on the (macro) process level by clustering programs to medium-grain sequential threads, running on parallel von Neumann nodes. On the (micro) instruction level, on the other hand, each partition of the dataflow program is translated to a thread in a sequential programming language. The exploitation of the remaining parallelism on an instruction level is then left to sequential language compilers, which are assumed to be capable of exploiting advanced processor features such as pipelining [PT91].

For sequential code generation for threads, the class of synchronous dataflow languages enjoys particularly efficient properties, yielding much superior performance characteristics for generated code as opposed to dataflow hardware [GEB03]. The multithreaded model fits well with current implementation techniques in domains such as automotive or avionics control systems, where highly optimized (and often hand-programmed) code for single threads, and communication buses between different processors, are nowadays standard.

For specification as opposed to programming, it has long been recognized that the dataflow approach, once the theoretical shortcomings of the Kahn model for specifying nondeterminism are overcome, is a powerful tool for specification as well [Bro86].

2.1.2 Why dataflow programs?

Simplicity of dataflow. Dataflow networks enjoy some typical properties of well-defined process concepts, such as commutative and associative composition, and recursiveness of the concept of a process, which again makes understanding and handling of designs simpler. Dataflow programs make control decisions *locally*, and in relation to this aspect, admit the *trace based formalization* based on Kahn's work. This choice has far-reaching implications both for easy understandability, and for powerful ways for methodical handling of designs, such as abstraction and refinement steps. We will explain each of these points in more detail.

Simple composition. Parallel composition of dataflow networks is commutative and associative: Composition imposes no particular order

($A||B = B||A$), and networks of a larger number of processes are unambiguously defined ($A||(B||C) = (A||B)||C = A||B||C$).

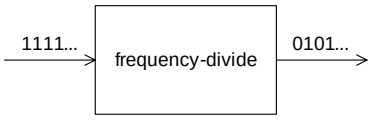
"Process" concept is recursive and unique. In dataflow programs, networks of processes are again processes, and processes are the sole unit of program structure. The simplicity is mirrored in the mathematical model, for instance defined by the semantics of SSDL in Section 2.2.2. The composition of many elementary functions in the form of an equation system is again a function from the (composite) inputs to the (composite) outputs.

Local control. A dataflow program has a clear distinction of inputs and outputs. Processes control only their outputs, and outputs are only controlled by their processes. Inputs are unconditionally accepted; composition in Kahn-like dataflow networks does not support the notion of a process "blocking" on an available input. Dataflow networks are therefore "input-enabled" [LT87]: they provide a response to any input[2]. This is a noteable difference to other *input-blocking* approaches to concurrency such as process calculi [Mil80][Hoa85], which do not strongly distinguish inputs and outputs, and thus allow processes to control a transaction symmetrically.

Simple trace-based formalization. Deterministic dataflow networks are fully characterized as functions over traces, or streams. As an example, consider the behavior of a frequency divider, shown in Fig. 2.1(a). The frequency divider receives a stream of `0`s and `1`s as inputs, forwards every second `1` as output, and outputs a `0` otherwise. As a deterministic dataflow program, the frequency divider can be fully characterized by a complete (and typically infinite) set of input/output pairs: Fig. 2.1(b) shows some example pairs. Trace-based formalization is comparatively simple and understandable, and admits lightweight definitions of abstraction and refinement relations between different stages of a design. For instance, if behavioral refinement is characterized by the subset relation on trace sets, then the important and intuitive class of trace-universal properties is preserved. Examples for such properties are "for all behaviors, a bad message $\bar{\sigma}$ is never produced" or "for all behaviors, a message σ is eventually followed by a message ϱ".

As a counterexample to dataflow, input-blocking formalisms do admit trace-based characterizations, but these characterizations are either

[2] In principle, this notion of response includes possibilities for under-specification, such as chaotic behaviors.



(a) Dataflow process

in	out
1	0
11	01
111	010
1111	0101
⋮	⋮

(b) Input/output pairs

Figure 2.1: A dataflow network example: `frequency-divide`

complex [Hoa85], or too abstract to fully capture the behavior. Resorting to more fine-grained observations of system runs, such as computation trees, can be problematic for handling refinement, or for compositional reasoning [LT87][BKP84].

Suitability of dataflow for software architecting. Languages in software engineering are used both for architecting software, and for programming software. Architecting here refers to large-grain design [PA92][SG96] of software, where the focus is chiefly on software components, their interfaces, connectors between components, and overall interconnection structure. While most dataflow languages are not specifically designed for coarse-grained definition of software, they share many characteristics with specific architecture definition languages (ADLs [MT00]), and are in many ways suitable for high-level design.

Simple concepts. Dataflow languages have only one (recursive) notion of process or component, and only one notion of communication by (possibly typed) dataflows. At first glance, this simplified universe of concepts may seem to stand in the way of "real" software design, which is inherently complex, and involves a plethora of heterogeneous notions of structure and communication. Consequently, other commercially established modeling techniques are typically much less homogeneous: In ASCET [ETA], there are at least four different classes of structural elements, and a multitude of communication mechanisms. A composition of ASCET processes is not a process, and the composition of ASCET methods is not a method. In UML [OMG05] designs targeted for real-time applications, classes and objects need to be classified as "active" or "passive" in accordance with their mapping to tasks in the implementation, which significantly determines their behavior. In UML-RT [IBM03][3], two "capsule" objects may communicate either

[3]We are using IBM's Rational RealTime tool here as a reference implementation of

by synchronous method calls, or by asynchronous messages.

There is, however, evidence that this richness of concepts is inadequate for describing software in an implementation-independent and coarse-grained way, such as in architecting. Having many different options for communication and composition can be problematic: they typically encourage developers to encode implementation-specific considerations in the model. For instance, a developer may choose to model a client/server-style interaction between two objects as a synchronous method call. This seemingly minor decision ties the two objects together for the implemententation; once the decision has been made, replacing the synchronous call by a communication primitive suitable for distribution is usually very hard and time-consuming. Similar problems have been reported by members of the Titus [E+97] project.

Trace-based characterization. There is some indication that for software engineering, trace- (or stream-) based characterizations are more understandable than tree-based characterizations. Vardi [Var01] concludes, in the context of temporal logics for model checking, that "it is simply much harder to reason about computation trees than about linear computations." The popularity of traces as a way to characterize reactive system is manifested by the widespread use of trace-like formalisms such as Message Sequence Charts [Int96], which support an example-oriented (existential) style of specification.

Explicit and complete notion of interface. In dataflow languages, similar to ADLs, the interface of the composite process is defined on the level of the composite in its entirety. Consequently, there are no "hidden" parts of the interface defined on the level of constituents further down in the containment hierarchy. In standard UML [OMG05], by comparison, the externally visible interface of a composite, consisting of a composite class aggregating other classes, may include publically visible attributes and operations of the aggregated classes, and therefore may be wider than the composite class's interface itself.

Explicit assignment of data to processes. The lack of implicit communication and side effects in dataflow and functional languages is an instance of data encapsulation. Data encapsulation is an important pre-

state machines and communication in the UML standard. This tool is both popular, and shares many characteristics with other real-time UML tools evaluated in a case study [SRS+03]. For aspects of the communication semantics, the current UML standard itself [OMG05] is too vague for an adequate assessment.

requisite for compositionality, which in turn is a crucial property for reliably building larger systems from smaller ones.

Order-independence. Because of commutative and associative composition, programs and specifications do not depend on the ordering or individual processes, which makes them easier to understand and less fragile in maintenance. Note that this is not true for the majority of commercially established modeling approaches: For instance, dataflow graphs in the popular ASCET-SD dataflow tool [ETA] need additional sequence numbers to be well-defined, and in implementations, ASCET processes must be assigned to tasks in a sequence, where `A; B;` may have different behavior than `B; A;`. Similarly, (multitasking) implementations of UML-RT need to incorporate scheduling choices in addition to the UML-RT model itself to obtain repeatable behavior, effectively yielding both non-commutative and non-associative composition.

Easy encoding as visual language. Associativity and commutativity allow for a straightforward encoding in a graph-like fashion, without additional annotation. Visual modelling has been a strong trend in software engineering recently, and the popular dataflow languages come with a graphical environment for specifying dataflow networks. Academically, research on the question whether visual languages are superior to textual languages, for instance with respect to to comprehensibility or overall productivity, is inconclusive. For instance, [BH95] claim that visual dataflow formalisms improve communication facilitated by visual syntax intuitive notation appealing to novices. [GP92], on the other hand, identify some problems and tradeoffs that occur when using visual dataflow languages vs. textual languages.

Programming: State of the art. In contrast to architecting, the term "programming" is interpreted here as the activity of fine-grained software design. We shall first take a critical look at the state of the art in the programming field before proceeding with an assessment of the suitability of the dataflow approach for programming.

From a platform point of view, von Neumann processors are the dominating platform for implementing software. Sequential languages, such as C/C++, Java, and Ada, are well-fitted to exploit the von Neumann processor. The languages' constructs for communication and control flow closely mirror the typical memory structure and available instruction set of the processor. As a consequence, in the fine-grained view from thread granularity down to the instruction level, software can be naturally expressed

in sequential languages. The close match of language and predominant architecture, along with the general familiarity of engineers with sequential languages, is probably the chief reason why sequential languages are so firmly established for fine-grained software design. The established concurrency primitives in programming, on the other hand, are strongly influenced by operating system (OS) concepts rooting back in the 1960s, such as threads, semaphores, monitors, and message pipes. We will refer to the combination of sequential threads and OS concurrency primitives as the *classic multithreaded* approach.

For programming concurrent systems, the classic multithreading approach still leaves things to be desired. Lee concludes in [Lee05] that multithreaded programs can be very difficult to understand, and that the use of threading primitives introduces a nontrivial amount of unrealiability to the programming process.

> **Example (Multithreaded Java application [Lee03]).** For illustrating the weaknesses of the classic multithreading approach, Lee [Lee03] cites a programming example from a large multithreaded Java application, where a single synchronization statement was added to otherwise highly reliable code. In the example application, the introduction of synchronization caused a potential deadlock, which went undetected in component testing, and was only accidentally discovered by users of the application. Analysis of the cause of the deadlock took weeks, and correction was difficult. Similar experiences with classic multithread programming has been recorded by multiple members of the AutoFOCUS project[HSE97].

The perceived incomprehensibility and unreliability in the domain of concurrent and embedded systems design may be tracked down to at least two conceptual misfits: *lack of compositionality*, and a missing *specification for timing*.

Lack of compositionality. The example of globally violating a property (deadlock-freedom) for a combination of locally validated and reliable subsystems is symptomatic for a more general phenomenon. For a large class of important properties, existing approaches for classic multithreading are not compositional. That is, if one obtains assurances individually about each component in a group, there is no systematic and transparent way to deduct assurances for the composition of individual components, except in trivial cases.

A typical manifestation of this problem is priority inversion [GS88]: pri-

ority inversion occurs when processes interact, for example by entering a monitor to exclusively access a shared resource. Under certain circumstances, the low priority process may block the high-priority process. The conceptual problem is that the apparent composition mechanism (priority-driven execution by a scheduler) is intercepted by a "hidden" and nonobvious form of interaction through the monitor mechanism provided by the operating system, leading to non-compositionality.

No specification for timing. For the embedded systems sector, time and the timing of behaviors is an indispensible part of the knowledge about an application and its interfaces to the exterior. In classic multithreading, knowledge about timing is not handled in a *specification* fashion, where times and time bounds are asserted beforehand and explicitly. Rather, time in classic multithreading is handled in a *programming* fashion, where timing properties can be inferred primarily on the level of atomic statements and primitives, thus involving detailed knowledge of the application software and platform. This is true both for *execution timing*, and for *activation rates*. We shall briefly summarize the problems with capturing timing through programming, as opposed to having a timing specification.

For execution timing, the only information inherently captured by a classic multithreaded design is the code itself, so the *actual* execution timing of a thread on a particular platform could be measured in lieu of a timing specification. However, compile-time worst case execution time (WCET) determination is notoriously difficult [KP02]. Therefore this way of capturing timing in the design is highly implicit, and hardly useful for establishing timing correctness for larger systems.

For activation rates, multithreading applications typically capture the activation policy of threads in low-level constructs. For instance, an interrupt service routine (ISR) called after the occurence of some external event may contain a call to the OS scheduler, which in turn releases an application task handling the event. As another example, timer interrupts may be used to increase internal counters, and timed application tasks are dispatched based on the value of these increasing counters. Again, for both examples, it is highly implicit how the different parts of multithreaded design are activated over time, how activations are causally related, in which order shared data is being processed by the different parts, and so on.

Suitability of dataflow for programming. Dataflow programming nicely resolves the issue of non-compositionality in programming, and provides a simple notion of timing specification based on timed interpretations of traces. However, in the field of programming, the prevailing dominance of sequential programming and the classic multithreading approach keeps the acceptance of dataflow models restricted to special domains. The two domains where dataflow programming is firmly established, though, are *signal processing*, and design of *digital controllers*. We will restrict our treatment to the latter, as the two domains share many characteristics.

In the digital controller field, synchronous dataflow tools such as MATLAB Simulink[4] or SCADE are well established. The assumption of a globally synchronized timebase in synchronous dataflow matches well with the uniform physical time assumptions inherent in controllers. Block diagrams for specifying controllers have been long established in control engineering education. Synchronous languages have successfully adapted concepts directed towards digital implementation, such as clock calculi (SIGNAL,LUSTRE) and sampling rates (SIMULINK), and concepts directed towards more heterogeneous and control-intensive designs, such as automata extensions.

For other domains, the potential and limits of dataflow programming (and, related, functional programming) are less clear, and the overall suitability in comparison to sequential, imperative programming is somewhat inconclusive. There are, however, some reports describing successful application of functional languages in other domains, such as reports describing successful experiences with the Erlang functional language in telecommunications applications [Arm96][Wig01].

2.1.3 Why synchronous?

Synchronous dataflow programs are a special case of *timed* dataflow programs. A dataflow program is timed if it admits a natural mapping from its message streams to a global timebase. Most often, this timebase is taken to be a discrete set. Two natural encodings for discrete time are explored, for instance, in [BS01]. The first encoding is to use special time tick symbols in streams to indicate the passage of one time unit. The second encoding is by directly associating the index position of a message in a stream with the global time.

At face value, timed dataflow allows natural modeling of *real time sys-*

[4]The discrete-time subset of Simulink qualifies as a synchronous language, as indicated by the direct translation of Simulink to other synchronous languages [CCM+03a].

tems, that is, systems sensitive to the passage of time in their environment. From a theoretical viewpoint, timed dataflow nicely resolves some issues in dataflow, as discussed below. However, there is no free lunch: the timed interpretation restricts dataflow processes to be *time-preserving*, that is, for each time unit's worth of messages consumed on the inputs, there must be one time unit produced as output. In the context of the SSDL language, a formal definition of a special case of time preservation is given in Section 2.2.2. In practice, time preservation forces consideration of time in the application model, something that comes in quite unnaturally in many cases, especially in early design.

Example (Natural untimed models). Consider a bank database, which sequentially processes incoming requests from a number of distributed ATMs. Let ✓ be the time tick symbol, and let σ be an incoming message. A timed specification must distinguish cases $\sigma\checkmark\sigma$ and $\sigma\sigma$. In the untimed model, both cases are equivalent. The specification for the untimed case is easier to obtain, and would typically be perceived as the more natural model.

As a serious drawback, though, the untimed interpretation of dataflow models suffers from two problems. The first problem is *resource boundedness*: The composition of two untimed bounded memory systems may not be bounded memory. This may be adequate for specification, but is certainly not acceptable for (embedded systems) programming. The second problem is that there is *no simple and abstract semantics* for nondeterminism. In order to be adequately expressive, untimed stream processing networks must allow for nondeterministic processes. The seemingly straightforward approach of extending the functional semantics of [Kah74] to relations over streams, however, is too abstract, and does not adequately capture the behavior of such processes. We will discuss each of these two points in some more detail.

Bounded resources. Information systems, and especially the kind of embedded systems targeted in this thesis, have only bounded resources (time, memory) to perform computations. From a theoretical viewpoint, untimed FIFO composition of individual bounded-memory processes without further constraints does not guarantee bounded resources for infinite executions. For an automata-theoretic argument, see [Cas92]. We shall illustrate this peculiarity of untimed FIFO composition with a small example:

Example (Unbounded integrator). Consider the dataflow process network `faulty-integrator` for a (Euler backward) integrator with faulty feedback loop, shown in Fig. 2.2. The designer of the integrator accidentally placed a `double` process in the loop: `double` forwards on its output two symbols for each symbol it receives on its input. We denote the *streams* processed by the individual processes as x, y, z, z', respectively. Each stream is a finite or infinite word of integer symbols.

Process	I/O behavior
`plus`	takes one symbol from each input and computes their sum as output Example: $\texttt{plus}(\texttt{123}\ldots, \texttt{123}\ldots) = \texttt{246}\ldots$
`double`	duplicates any incoming symbol on its output Example: $\texttt{double}(\texttt{123}\ldots) = \texttt{112233}\ldots$
`fby0`	outputs `0` as first symbol, followed by the incoming symbols Example: $\texttt{fby0}(\texttt{123}\ldots) = \texttt{0123}\ldots$

Process `faulty-integrator` is composed from operators that each in isolation require only bounded memory. Using the time encoding based on message indices, though, we can clearly deduce that `double` is not time-preserving. Consequently, `faulty-integrator` is an instance of the more general case of untimed process networks, where the composition may require unbounded memory.

It shows that the example is indeed not bounded-memory. Because `plus` processes x and z' symbols synchronously, and because `double` produces two z symbols for each incoming y symbol, there will be unprocessed z or z' symbols in the feedback loop for any nonempty run. If we use a length operator $\#$ to measure the length of a stream, then it is easy to show that the memory required for storing these unprocessed symbols is at least $\#y$ symbols. It is also clear that, for any given finite memory bound $k > 0$ for the symbol storage, there is always a run of `faulty-integrator` where the symbol storage overruns k.

Popular modeling formalisms such as UML-RT [IBM03] also use the untimed form of composition: messages are not time stamped, and processes are not restricted to be time-preserving. Consequently, there are no static guarantees about resource consumption, and it is easy to construct state machines that will eventually overrun any memory limit. Clearly, there is no easy way to provide bounds on timing in UML-RT either: in an industrial case study Dohmen and Somers [DS02] conclude that "UML-RT does not support the [...] implementation and verification of hard real-time

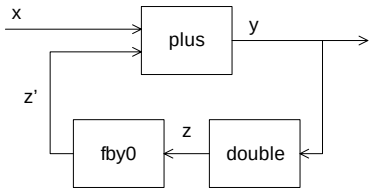


Figure 2.2: An unbounded dataflow network: `faulty-integrator`

constraints." There are, of course, a number of academic contributions for real-time analysis of UML models, such as [SPFR98], which typically rely on a fixed mapping of capsules/objects to tasks, and a fixed assignment of task priorities. However, the additional mapping information is typically not considered part of the UML-RT "model" itself, as it would severely limit implementation-independence and portability of the design.

No simple and abstract semantics. The deterministic dataflow semantics of Kahn [Kah74] is simple, and admits the well-defined semantic framework of stream-processing functions. However, Kahn networks without time assumptions are generally not expressive enough. For instance, various flavors of "fair merge" operators are characterized by the fact that they can (operationally) test for absence. Testing for absence is essentially a distinction in the input/output behavior based on relative *timing* of message sequences, and cannot be captured by untimed streams of data. Therefore, we clearly have nondeterminism on the level of the untimed input/output semantics.

Though this may sound like a fairly theoretical argument, time-sensitive operations such as fair merging are very important for practical applications.

> **Example (Fair merge).** Consider (1) again the bank database fusing incoming ATM requests, (2) a vehicle dynamics coordinator in automotive applications, which merges driver requests (accelerate, brake) with stability intervention requests (e.g. car is skidding) in real time. Both applications require merging of data in a fair, timely, and fault-tolerant manner.

The straightforward approach to extend the Kahn model towards nondeterminism is to use history *relations* over streams. Due to the level of detail of this semantics, however, the composition of some context with two semantically equivalent non-deterministic dataflow systems, respectively, may not yield two equivalent systems. The relational approach to

nondeterministic dataflow networks is therefore non-compositional. This is the Brock-Ackermann anomaly described in [BA81]. Some further discussion of this phenomenon will be given in the appendix.

2.1.4 Why discrete?

Time domains and computational power. By the way digital computers are built, they carry with them a notion of discreteness of computations. This is most clearly visible at the level of processor, or bus, cycles. Above this level, it is also quite clear that basic instructions and bus cycles are grouped to form more complex operations, such as the one-time execution of a task, or a bus transaction. Thus, computational steps at any level of detail are performed discretely, and the accurate handling of environment events by a computer system is restricted to environments where some minimum event separation holds. The straightforward solution to model discrete computational steps is by choosing a discrete timebase, such as the natural numbers, to logically model computational progress.

From the application point of view, this simple way of modeling time may be inadequate. For instance, continuous controllers are frequently encountered in the embedded systems domain: these systems are most naturally modeled with respect to a *dense timebase*. Moreover, modelling continuous mathematics theoretically requires an *infinite* number of computations in a given time interval [Sta01]. In practice, symbolic and/or numerical analysis with a finite number of computations is used, and continuous-time design environments [MW] essentially "hide" the finiteness of the computer by approximating high-level symbolic constructs, such as a continuous-time integrator, by means of a finite numerical integration algorithm.

Certain applications in discrete control, such as automata with real-valued delays [AD94], form yet another application category with a *dense timebase*, but *finitely* many computations. The finiteness of computations is typically captured by some temporal progress criterion. This category is also often referred to by the term "discrete-event systems".

Under these circumstances, the choice of an adequate model for time vs. computations becomes less clear. Let us summarize the two dimensions of computational power and time domain in the context of reactive systems:

Finite vs. infinite number of iterations. In a given finite time interval, is the number of possible iterations finite or infinite?

Real vs. logical time domain. Should the time domain, i. e. the set from which timestamps are drawn, correspond to real time (usually taken as a dense set), or should the time domain be discrete, and therefore purely logical?

The first question is relatively easy to answer in the context of this thesis: being oriented toward design and implementation, and having to cope with finite computational resources, we restrict ourselves to the finite-computation model, which can be directly realized with finite computational power without having to resort to numerical techniques, provided that iterations themselves are bounded.

Though this kind of restriction is frequent in control design tools, and continuous-time modeling is generally being challenged by implementation concerns [Eis05], one should by no means deny the importance of continuous-time modeling. For instance, for closed-loop controller design, where the controller interacts with continuous-time physics, continuous-time controllers are easier to describe than discrete-time controllers, and are much more amenable to stability analysis.

To integrate continuous-time and discrete-time modeling in a development process, related work [AGLS01][Sta01] provides precise methods to refine continuous-time specifications into discrete-time ones.

The second question, whether to use a real or logical time domain, is more involved, and may be examined with respect to *formal analysis*, suitability for *distributed implementation* in networks, and abstraction from *nondeterministic implementation effects*.

Formal analysis. For formal analysis, it shows that many of these primary validation and observation tasks of interest are much more easily performed on discrete-time abstractions than dense-time abstractions. For instance, most model checking problems are at least PSPACE-hard over Timed Automata [AD94], which is a popular dense-time formalism in verification. In comparison, many model checking problems for discrete time are polynomial. For formalizing problems in formal verification, discrete-time formalisms are quite expressive: one can capture a very significant share of dense-time verification problems as discrete-time problems, for instance by a digitization procedure [HMP92].

Distributed implementation. When distributing applications across networks, typical failures and uncertainties such as transfer delays, node failures, and transmission failures have to be considered. At the same time, *agreement problems* occur frequently, such as a common operational mode shared between different nodes, or a consensus between

replicated nodes. Agreement problems in the presence of failures and uncertainties are known to be hard in the asynchronous case, where no common time grid and discrete step duration is defined. They can be much more efficiently handled if networks are synchronized, and a common discrete timebase (corresponding to a discrete timebase in an application model) is used across the network [Kop92].

Nondeterministic implementation effects. Most real-time HW/SW systems have more or less nondeterministic timing behavior at a small time granularity. Reasons range from highly history-dependent processor performance due to pipelining and caching [KP02] to large jitters and unpredictable service caused by dynamically arbitrated communication media (e.g. CAN) [BBRN05]. These effects can, to some extent, be countered by coarse timing assumptions and dynamic implementation strategies. Firstly, if the time grid used for observations is coarse, such as "execute the task every 20ms, with a deadline of 20ms", the timing jitters of individual instructions and message transfers may cancel each other out in the best case. Secondly, dynamic implementation strategies such as preemptive and priority-driven scheduling, if correctly configured, can ensure high probabilities for timely execution of high-priority tasks, effectively providing a quasi-deterministic platform for the high-priority segment of an application. Consequently, the combination of discrete, coarse-grained time modeling and dynamic scheduling has been explored recently by numerous researchers [HHK01][SC04][BR04].

To summarize this section, dataflow programming is a well-established paradigm which invariantly offers a number of benefits with respect to simplicity, suitability for architecting software, and suitability for programming, over some more established techniques. Although these merits have been known and advertised for a long time, the current complexity challenge for distributed embedded software systems may shine a new light on dataflow programming, particularly for the problem of software architecting. As a special case of dataflow, the timed and time-synchronous dataflow approaches address more particular issue such as real-time systems modeling, simple and adequate semantic models, and resource-boundedness guarantees. In the following section, we shall define a (time-) synchronous dataflow language illustrating this paradigm.

2.2 The synchronous dataflow language SSDL

The remaining chapters of this thesis will deal with the illustration of a synchronous dataflow languages reminiscent of other languages like LUSTRE, SIGNAL, and AutoFOCUS. For this purpose, we define a very simple synchronous dataflow language called SSDL (Simple Synchronous Dataflow Language). SSDL uses an equational form of specifying dataflow programs, which can be easily derived from a graphical box-and-arrow formalism, as in AutoFOCUS, discrete-time SIMULINK, or SCADE. SSDL deliberately avoids some more complex language issues such as *data types*, or *non-periodic clocks* and *clock checking*. Clocks are generally used to carry frequency information, and to check well-formedness of programs with respect to presence/absence of values.

From a practical usability perspective, these features are all very important for a synchronous language. However, for the purpose of illustrating the semantics of synchronous programs, for working with examples, and for formalizing linearizations of synchronous programs in Chapter 4, SSDL comprises all necessary features and shall suffice as a simple formalization. We will start out with a reduced language, *Mini-SSDL*, which lacks primitives for composition of programs, and for easily specifying multirate programs. The remaining SSDL primitives shall be introduced in Section 2.2.3.

2.2.1 Mini-SSDL syntax

Programs. A Mini-SSDL program P is a system of equations over variables $\mathtt{x_i} \in X, 1 \leq i \leq n$. X is the *variable set* associated with P. Variables are used for communication between processes.

Definition 2.1 (Mini-SSDL program syntax). *A Mini-SSDL program has the following principal structure* ($1 \leq m \leq n$)*:*

$$
\begin{array}{l}
\texttt{program P;} \\
\quad \texttt{input} \quad \mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_l; \\
\quad \texttt{var} \quad \mathtt{x}_{l+1}, \mathtt{x}_{l+2}, \ldots, \mathtt{x}_m; \\
\quad \texttt{output} \quad \mathtt{x}_{m+1}, \mathtt{x}_{m+2}, \ldots, \mathtt{x}_n; \\
\\
\quad \mathtt{x}_{l+1} := \mathtt{e}_{l+1}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n); \\
\quad \mathtt{x}_{l+2} := \mathtt{e}_{l+2}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n); \\
\quad \vdots \quad \vdots \quad \vdots \quad \ddots \quad \vdots \\
\quad \mathtt{x}_n := \mathtt{e}_n(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n); \\
\texttt{endprogram;}
\end{array}
$$

Variables $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_l$ are declared as *input variables* $X_I \subseteq X$ of the program using the `input` keyword, and have no defining equations. The run-time value of an input variable is provided by the environment. All other variables $\mathtt{x}_{l+1}, \mathtt{x}_{l+2}, \ldots, \mathtt{x}_n$ have defining equations, and are declared as either *local variables* $X_L \subseteq X$, using the `var` keyword, or as *output variables* $X_O \subseteq X$, using the `output` keyword for declaration. Input and output variables together define the externally visible behaviors of the program, while local variables are hidden from the exterior. The difference between visible and hidden variables will become apparent in the SSDL program composition described in Section 2.2.3. We will sometimes refer to variable tuples in a compact form instead of individual variables, written $\bar{\mathtt{x}} = (\mathtt{x}_\mathtt{i}, \mathtt{x}_\mathtt{j}, \ldots)$.

No particular order is enforced on the declaring equations. In fact, associativity and commutativity allow any rearrangement without changing the program's behavior.

Expressions $\mathtt{e} \in E$ in Mini-SSDL are built up from a combination of Mini-SSDL *operators*, *constants* $\mathtt{c} \in C$, and variables $\mathtt{x} \in X$. In a Mini-SSDL program, the right-hand side expressions $\mathtt{e}_i$ are used to define the computation for variables $\mathtt{x}_i$, respectively. Possible Mini-SSDL constants that may appear in expressions are:

Boolean constants: `True`, `False`

Integer constants: $\ldots, -1, 0, 1, \ldots$

Undefined constant: `Nil`

The operators of Mini-SSDL used for building expressions are:

Arithmetic operators such as addition, subtraction, multiplication, and division

Comparison operators such as equality, inequality, greater, less,

Logical connectives such as negation, conjunction, and disjunction,

Choice such as the `if.then.else.fi` statement, and

Delay such as the `fby` statement. This operator will be briefly motivated below.

Tupling and brackets for forming the tuple of individual expressions

More formally, expressions $\texttt{e}, \texttt{eqn}, \texttt{eqns} \in E$ are defined as, for $\texttt{c} \in C$, $\texttt{x} \in X$, $\texttt{e}_1, \texttt{e}_2, \texttt{e}_3 \in E$,

$$
\begin{aligned}
\texttt{e} \;&::=\; \texttt{c} \mid \texttt{x} \mid (\texttt{e}_1 \big[, \texttt{e}_2\big]^*) \mid \texttt{e}_1 + \texttt{e}_2 \mid \texttt{e}_1 - \texttt{e}_2 \mid \texttt{e}_1 * \texttt{e}_2 \mid \texttt{e}_1 \,/\, \texttt{e}_2 \mid \\
&\quad\; \texttt{e}_1 = \texttt{e}_2 \mid \texttt{e}_1 <> \texttt{e}_2 \mid \texttt{e}_1 < \texttt{e}_2 \mid \texttt{e}_1 > \texttt{e}_2 \mid \texttt{not}\, \texttt{e}_1 \mid \\
&\quad\; \texttt{e}_1\, \texttt{and}\, \texttt{e}_2 \mid \texttt{e}_1\, \texttt{or}\, \texttt{e}_2 \mid \texttt{if}\, \texttt{e}_1\, \texttt{then}\, \texttt{e}_2\, \texttt{else}\, \texttt{e}_3\, \texttt{fi} \mid \texttt{e}_1\, \texttt{fby}\, \texttt{e}_2 \\
\texttt{eqn} \;&::=\; \texttt{x} := \texttt{e} \\
\texttt{eqns} \;&::=\; \big[\texttt{eqn};\big]^+
\end{aligned}
$$

The delay operator `fby` deserves further explanation, as it is typically not found in non-dataflow languages. In order to be adequately expressive for stateful designs, dataflow languages need to allow feedback, such as the feedback loop illustrated for the integrator in Fig. 2.2. But according to the causality analysis defined in Section 2.2.4, the only acceptable form of feedback for SSDL programs is *delayed* feedback: programs like $\texttt{x} := \texttt{x}$ with immediate feedback are rejected. Instead, one needs an operator that expresses the fact that the "last" value of `x` is to be used instead of the current value.

As illustrated in Chapter 3, the delay operator is required in the language for yet another purpose: because the timebase of a synchronous dataflow program serves as an abstraction of actual delays and computation times in implementation, synchronous dataflow programs sometimes need discrete delays at certain partitioning boundaries in order to be implementable with given computational resources.

Note that, SSDL being an untyped language, we are not particular about the comprehensibility or conceptual soundness of expressions. For instance, `if 1 + 1 then True else 1 fi` is a valid expression in SSDL. Expressions in a program P are, however, restricted so that P is a *reactive* program: a detailed account of the resulting restrictions on the level of syntax is given in Section 2.2.4.

Example (Frequency divider). The Mini-SSDL program `frequency-divide` shown in Fig. 2.3 is a synchronous realization of the frequency divider previously described in Fig. 2.1: it receives a tick signal as input, and forwards every second tick as its output. The variables `in`, `laststate`, `state`, and `out` are of integer type: a value of `1` denotes a time tick, while `0` denotes the absence of a tick.

An example run of `frequency-divide` is as follows:

```
program  frequency-divide;
  input    in;
  var      state, laststate;
  output   out;

  state     := if in=1 then 1-laststate else laststate fi;
  laststate := 1 fby state;
  out       := if in=1 then state else 0 fi;
endprogram;
```

Figure 2.3: Program `frequency-divide`

step	1	2	3	4	5	6	7	...
in	0	1	1	0	1	0	1	...
laststate	1	1	0	1	1	0	0	...
state	1	0	1	1	0	0	1	...
out	0	0	1	0	0	0	1	...

We observe that, for instance in step 3, `in`'s value influences the value of `out` in the same step, even though the consumption of `in` causally precedes the production of `out`. This kind of instantaneous communication is a hallmark of synchronous languages (sometimes called "synchrony hypothesis"), or fixed point semantics for timed systems in general.

2.2.2 Mini-SSDL semantics

Streams. The semantic representation of a variable's evolution is a *stream*. A stream is a finite or infinite sequence of *symbols* from a symbol set Σ. For the purpose of defining SSDL, Σ is simply taken to be the set of values taken by individual variables, V. Values may include Boolean values, where we shall use tt to indicate a "true" Boolean value, and ff for "false". For such a variable set, we use $V^\omega = V^* \cup V^\infty$ to denote the set of all finite and infinite streams over V[5]. Consequently, we denote the stream for an SSDL variable `x` by $x \in V^\omega$. For a stream $w \in V^\omega$, the i-th symbol is written as w_i, starting at w_1 for the first symbol.

The *length operator* $\#$ yields the length of the stream to which it is applied. It can be extended to stream tuples, yielding a tuple of lengths.

[5]We follow the convention in e. g. [BS01] of denoting infinite streams with ∞, and the union of finite and infinite streams with ω. Most works e. g. from the formal verification community use the opposite notation: infinite streams are denoted with ω, the union of finite and infinite streams with ∞.

Concatenation of streams, written $w_1 . w_2$, yields a stream that starts with the messages of w_1 followed by the messages of w_2. The *empty stream* is denoted as ε: it is the neutral element of concatenation, so for any stream w, $w . \varepsilon = \varepsilon . w = w$. The *projection operator* $|_{(.)}$ is used to project away messages: $w|_{V'}$ is the substream of w obtained by removing all messages in w that are not in the set $V' \subseteq V$.

A stream w is a *prefix* of another stream w', written $w \sqsubseteq w'$, if there exists some (possibly empty) stream w'' such that $w' = w . w''$. The pointwise extension of constants and functions to streams in the following shall be necessary to define the semantics of SSDL.

Definition 2.2 (Pointwise extension to streams). *The pointwise extension of a constant c to a stream, written c^ω is the stream obtained by infinite repetition of c. Similarly, the pointwise extension of a function $f : V \to V'$ to a function $f^\omega : V^\omega \to (V')^\omega$ over streams is given by the following inductive definition:*

$$\begin{aligned} f^\omega(\varepsilon) &= \varepsilon \\ f^\omega(w . \sigma) &= f^\omega(w) . f(\sigma) \text{ for all } \sigma \in V, w \in V^\omega \end{aligned}$$

Semantics map and metalanguage. We shall use a *semantics map* to map SSDL programs and their parts to their mathematical equivalent, or denotation. Obviously, the denotation of a constant (which will invariantly map to the same value) is a somewhat different structure than the denotation of an expression or variable (which is rather a function from some given environment to a value), or a program (which maps tuples of environments to tuples of values). In order to simplify notation, we shall define our semantics based on a "metalanguage" L_ρ using:

Constant streams such as tt^ω.

Variable environments $\rho(\mathtt{x})$, where the environment map ρ maps variables $\mathtt{x}, \mathtt{y}, \mathtt{z}, \ldots$ to their corresponding streams. We denote by $Dom(\rho)$ the domain of ρ. The environment ρ may be updated for a syntactic variable $\mathtt{x}$ by a semantic equivalent y according to

$$\rho[\mathtt{x}/y](\mathtt{z}) = \begin{cases} \rho(\mathtt{z}) & \text{if } \mathtt{z} \neq \mathtt{x} \\ y & \text{if } \mathtt{z} = \mathtt{x} \end{cases} .$$

Lambda notation for expressing function abstraction $\lambda x.e$ and function application $e(x)$.

Tupling operator for expressing grouping of elements to tuples (e_1, e_2). We shall use an alternative compact notation for tuples: for instance, an n-tuple of stream variables, $(x_i, x_j, \ldots)$ can alternatively be written as $\bar{x}$.

Least fixed point operator for expressing least fixed points of an expression with respect to the prefix order $\sqsubseteq$ on streams, $\mathrm{lfp}\, e$. The appendix describes this operator, and the properties of the prefix order, in more detail.

We ensure that the metalanguage, again, is based on well-founded mathematical constructs [Win93]. Based on the definition of the metalanguage, we are ready to define a *semantics map* to map expressions to their denotational counterparts.

Definition 2.3 (Semantics map). *Let $\rho : X \to V^\omega$ be a variable environment, and let L_ρ be the metalanguage associated with ρ. The semantics map $[\![.]\!]_\rho : E \to L_\rho$ maps expressions $\mathtt{e} \in E$ to a metalanguage term $l \in L_\rho$.*

In later parts of this thesis, we may drop the subscript and write $[\![.]\!]$ instead of $[\![.]\!]_\rho$ if the nature of ρ is not an essential part of the description. We shall use several notational conventions in order to define the semantics. For a program

$$\mathtt{program}\,\mathtt{P};\ \mathtt{input}\,\bar{\mathtt{x}};\ \mathtt{var}\,\bar{\mathtt{y}};\ \mathtt{output}\,\bar{\mathtt{z}};\ \mathtt{eqns};\mathtt{endprogram};,$$

where `eqns` are the program's equations, we write $\mathtt{P}(\bar{\mathtt{x}});\mathtt{eqns};$ as a shorthand. Operators in general are referred to as `op` and op in the syntax and semantics domain, respectively. The operator `if.then.else.fi` is abbreviated in prefix form as `ite`$(.,.,.)$. The semantics of Mini-SSDL is then defined in Fig. 2.4. Some additional comments:

- In the definition of `fby`, $([\![\mathtt{e}_1]\!]_\rho)_1$ denotes the first symbol of the stream for expression $\mathtt{e}_1$. The first argument of `fby`, e_1, is typically a constant value, so taking the first symbol of its stream extension is equivalent to simply using its constant (element) value.

- The stream-extended functions for arithmetic, comparison, and logical operators are standard, yielding the undefined value `Nil` as a result for all undefined cases (e. g. `1 or False`, `1 + False`). For `if.then.else.fi`, we define an element function $\mathrm{ite} : V \times V \times V \to V$ as

$$\mathrm{ite}(x, y, z) = \begin{cases} y & \text{if } x = tt \\ z & \text{otherwise} \end{cases} \quad \text{for } x, y, z \in V,$$

  and use the stream-extended function $\mathrm{ite}^\omega : V^\omega \times V^\omega \times V^\omega \to V^\omega$ for the semantics definition of `ite`.

- The semantics of programs, $[\![\mathtt{P}(\bar{\mathtt{x}});\mathtt{eqns};]\!]_\rho$, is associated with a variable substitution $[\bar{\mathtt{x}}/\bar{y}]$, where $\bar{y}$ is a "fresh" variable tuple not in $Dom(\rho)$. The rationale behind this substitution will become apparent in Section 2.2.3, where programs P can be instantiated potentially several times such that each program instantiation operates on a distinct stream tuple in the semantics.

Constants			
$[\![\mathtt{True}]\!]_\rho$	$=$	tt^ω	
$[\![\mathtt{False}]\!]_\rho$	$=$	ff^ω	
$[\![n]\!]_\rho$	$=$	n^ω	for $n \in \mathbb{Z}$
$[\![\mathtt{Nil}]\!]_\rho$	$=$	$\bot^\omega$	
Variables			
$[\![\mathtt{x}]\!]_\rho$	$=$	$\rho(\mathtt{x})$	
Operators			
$[\![\mathtt{op(e)}]\!]_\rho$	$=$	$op^\omega([\![\mathtt{e}]\!]_\rho)$	for $\mathtt{e} \in E$, $\mathtt{op} \in \{+,-,*,/,=,<>,<,>,\mathtt{not},\mathtt{and},\mathtt{or},\mathtt{ite}\}$
$[\![\mathtt{e}_1\ \mathtt{fby}\ \mathtt{e}_2]\!]_\rho$	$=$	$([\![\mathtt{e}_1]\!]_\rho)_1 \fatsemi [\![\mathtt{e}_2]\!]_\rho$	
Equations			
$[\![\bar{\mathtt{x}} := \mathtt{e}]\!]_\rho$	$=$	$\rho\left[\bar{\mathtt{x}}/\left(\mathrm{lfp}\ \lambda\bar{y}.[\![\mathtt{e}]\!]_{\rho[\bar{\mathtt{x}}/\bar{y}]}\right)\right](\bar{\mathtt{x}})$	for $\bar{y} \cap Dom(\rho) = \emptyset$
Tupling			
$[\![(\mathtt{e}_1,\mathtt{e}_2)]\!]_\rho$	$=$	$([\![\mathtt{e}_1]\!]_\rho, [\![\mathtt{e}_2]\!]_\rho)$	
$[\![\mathtt{eqn}_1;\mathtt{eqn}_2;]\!]_\rho$	$=$	$([\![\mathtt{eqn}_1]\!]_\rho, [\![\mathtt{eqn}_2]\!]_\rho)$	
Programs			
$[\![\mathtt{P}(\bar{\mathtt{x}});\mathtt{eqns};]\!]_\rho$	$=$	$\lambda\bar{y}.[\![\mathtt{eqns}]\!]_{\rho[\bar{\mathtt{x}}/\bar{y}]}$	for $\bar{y} \cap Dom(\rho) = \emptyset$

Figure 2.4: Semantics of Mini-SSDL

The remainder of this section will be concerned with some of the mathematical theory behind the denotational definition of synchronous dataflow programs, and will in particular demonstrate the existence of a least fixed point in the semantics.

Sets of streams as complete partial orders. The prefix relation $\sqsubseteq$ over streams is a partial order: it is antisymmetric ($w \sqsubseteq w' \wedge w \sqsupseteq w' \Rightarrow w = w'$),

reflexive ($w \sqsubseteq w$), and transitive ($w \sqsubseteq w' \wedge w' \sqsubseteq w'' \Rightarrow w \sqsubseteq w''$). The partial order has a unique bottom element, ε. It is also a *complete* partial order, and induces a least upper bound operator $\bigsqcup$ (Defs. A.1–A.3 in the appendix).

Synchronous input/output functions are required to have a particular property called *length preservation*. Length preservation induces that there is a fixed correspondence between lengths of input streams $x_1, \ldots, x_n$ and output stream x_i. This reflects the uniform, parallel progressing of time in synchronous system: A process will produce one additional symbol on its output stream when all input streams are extended by one additional symbol.

Definition 2.4 (Length preservation). *Let $\bar{x} \in (V^*)^n$ be an n-tuple of streams, and let $\#\bar{x} \in \mathbb{N}^n$ be the tuple denoting the lengths of its component streams. Let f be a function mapping $\bar{x}$ to a stream $y \in V^*$. Then f is length-preserving iff*

$$\exists \bar{k} \in \mathbb{N}_0^n \,.\, \forall \bar{x} \in (V^*)^n \,.\, \#f(\bar{x}) = \min(\#\bar{x} + \bar{k}),$$

where $\#\bar{x} + \bar{k}$ denotes the component-wise addition of $\#\bar{x}$ and $\bar{k}$, and min *identifies the smallest component of the tuple.*

Length preservation says something about the relative length of streams, but does not allow any narrow conclusion about the relation of prefixes of inputs and outputs. Consider the function yielding stream `1` for all inputs of length one, stream `2.2` for all inputs of length two, stream `3.3.3` for all inputs of length three, and so on. This function is certainly length-preserving, but defies our intuition of a causal and stepwisely operating computing system.

In dataflow networks, an intuitive property stemming from causality is that, given a function f over streams, f will at most append additional symbols to its output given additional input, but never replace or remove past output symbols. That is, given some history of inputs x_1 with output history $f(x_1)$, an extension of x_1 to some x_2 such that $x_1 \sqsubseteq x_2$, will result in an output history $f(x_2)$ such that $f(x_1) \sqsubseteq f(x_2)$. On the level of semantics, this property corresponds to *monotony* of stream processing functions. It is also an important prerequisite for the existence of least fixed points for dataflow networks. The following two definitions are directly related to the existence of least fixed points for functions.

Definition 2.5 ((Scott) monotony). *A function f from an n-tuple of streams $\bar{x} = (x_1, x_2, \ldots, x_n)$ to a stream y, $\bar{x} \in (V^\omega)^n, y \in V^\omega$, is (Scott) monotonic iff*

$$\forall \bar{x_1} \in (V^\omega)^n \,.\, \forall \bar{x_2} \in (V^\omega)^n \,.\, \bar{x_1} \sqsubseteq \bar{x_2} \Longrightarrow f(\bar{x_1}) \sqsubseteq f(\bar{x_2})$$

As a corollary, we note that length preservation and monotony combined yields prefix-closure of the function's image (Lemma A.5 in the appendix). A second important property, which encompasses monotony, is *continuity*. In a sense, continuity forces that a function is both monotonic, and that its behavior is fully described by its behavior for finite inputs.

Definition 2.6 ((Scott) continuity). *A function f from an n-tuple of streams $\bar{x} = (x_1, x_2, \ldots, x_n)$ to a stream y, $\bar{x} \in (V^\omega)^n, y \in V^\omega$, is (Scott) continuous iff*

1. *f is monotonic.*
2. *For all chains $\bar{x_1} \sqsubseteq \bar{x_2} \sqsubseteq \cdots \sqsubseteq \bar{x_i} \sqsubseteq \cdots$, it holds that*

$$\bigsqcup f(\bar{x_i}) = f\left(\bigsqcup \bar{x_i}\right)$$

Based on the element function and their pointwise extension to streams, we can easily establish the following properties on the level of elementary SSDL operators.

Property 2.1. *For all elementary Mini-SSDL operators, the corresponding function on the level of semantics is length-preserving.*

Property 2.2. *For all elementary Mini-SSDL operators, the corresponding function on the level of semantics is continuous.*

The semantics gives meaning to expressions built from the elementary operators. Consequently, we have to ensure length-preservation and continuity for expressions, and therefore programs, as well. Fortunately, it can be shown that all operators of the metalanguage preserve length-preservation and continuity [Win93]. So we end up with two further results:

Property 2.3. *For all Mini-SSDL expressions, the corresponding function on the level of semantics is length-preserving.*

Property 2.4. *For all Mini-SSDL expressions, the corresponding function on the level of semantics is continuous.*

Based on Property 2.4, it is possible to show that the least fixed point for any program P is defined and unique. *Kleene's fixed point theorem* shows that, for a continuous function f, the least fixed point is the upper bound of a the ω-chain obtained by repeated application of f.

Theorem 2.5 (Fixed-point theorem (Kleene)). *Let $f : D \to D$ be a continuous function on a cpo $(D, \sqsubseteq)$ with bottom $\bot$. Define as $f^j(x)$ the j-fold application of function f to element $d \in D$. Define*

$$\mathrm{lfp}(f) = \bigsqcup \{f^j(\bot) \mid j \in \omega\}.$$

Then $\mathrm{lfp}(f)$ is the least fixed point of f on $(D, \sqsubseteq)$.

Consequently, the fixed-point theorem gives an effective procedure for constructing the least fixed point of the equation system for a Mini-SSDL program P, as we know that both individual functions f_i are continuous, and that the tuple function $\bar{f}$ is also continuous.

2.2.3 SSDL

In the definition of Mini-SSDL in Section 2.2, we have not included the possibility of composing *programs* with each other, only operators and equations. We shall see that one can easily overcome this limitation: SSDL extends Mini-SSDL by composition of programs, which shall be helpful when talking about implementation schemes for larger software structures (superprograms, subprograms) in Chapters 3 and 4. For multirate composition, SSDL introduces an additional `every` primitive, which alters the behavior of its enclosed operators and subprograms.

Composition of programs. According to the Mini-SSDL semantics, the behavior of a program is defined as a function mapping (tuples of) input streams to (tuples of) output streams. We have also seen how functions (operators) may be mutually composed on the level of equations. Because programs are simply functions, it is easy to extend SSDL for composing programs. This simplicity is a direct consequence of the "uniqueness of process concept" principle introduced in the discussion of dataflow programs in Section 2.1.2.

Together with tuple constructor primitives, this enables us to include (sub-) programs in expressions just like regular operators, and to use a superordinate program to express the composition of subprograms. Before defining subprogram composition formally, we shall illustrate it with a small example.

Example (Declaring and using the `hold` operator.). The `hold` operator is a common primitive that is used to translate between streams of different frequencies. Such streams of different frequencies can be

expressed in SSDL by interleaving regular (Boolean, integer) symbols with `Nil` symbols. `hold` always yields the last non-`Nil` symbol of its incoming stream.

We declare and use the `hold` operator in an example program, `threeish`, shown in Fig. 2.5. `threeish` uses a sub-program, `every-three`, which creates a `Nil`-interleaved sequence of successive multiples of 3. `every-three` is then combined with `hold` to replace `Nil` symbols with the respective last value.

Note that sub-programs are referenced as `every-three` and `hold(.)` on the right-hand side of the defining equation for `out` of program `threeish`. The call to `every-three` has no arguments (inputs) and one output, while the call to `hold` has one input and one output. The output is assigned to output `out` of program `threeish`. A run of `threeish` yields the following sequence:

step	1	2	3	4	5	6	7	...
every-three	0	Nil	Nil	3	Nil	Nil	6	...
hold(every-three)	0	0	0	3	3	3	6	...

```
program threeish;
  output out;

  program every-three;
    var state;
    output out;

    state := 0 fby state+1;
    out := if (state/3)*3 = state then state else Nil fi;
  endprogram;

  program hold;
    input in;
    output out;

    out := if in<>Nil then in else Nil fby out fi;
  endprogram;

  out := hold(every-three);
endprogram;
```

Figure 2.5: Program `threeish`

The `every` operator. In the above example, it was somewhat clumsy to specify the sub-program `every-three`. In fact, `every-three` seems to be an instance of the simpler program defined by the equation

```
out := 0 fby out+3;
```

The only difference between the program defined by the above equation and `every-three` is that `every-three` is computed at a "slower" rate, with two `Nil` symbols interleaved between successive values of `out`. The `every` operator is used in SSDL to compute expressions or subprograms at such a slower rate: it takes an expression as its first argument, and an integer value as its second argument. We shall call this second argument *clock* in the sequel. For clock n, successive output values of some `e every n` will be interleaved by $n - 1$ occurences of `Nil`. Clearly, the clock is required to be greater or equal to 1. It follows that for any expression `e` $\in E$, `e every 1` is equivalent to `e`.

Using the `every` operator, we have a simpler way of writing `every-three`: the equation

```
out := 0 fby out+3 every 3;
```

yields equivalent behavior. Note that unlike the other SSDL operators, `every` is not a first-order operator, as it modifies the expression or program it is applied to, rather than simply operating on its outputs. Short of defining SSDL as a general higher-order synchronous language [CP96], we shall use a simple set of rewrite rules (Fig. 2.8), which translate any `every`-annotated program to an `every`-free equivalent. The translation is based on two macro operators, `sample` and `fby-every`, which are defined by the programs in Figs. 2.6 and 2.7, respectively. Furthermore, for defining program composition in SSDL, the set op of possible operators is extended by the set of subprogram references defined in the current scope, SP. We assume absence of recursive subprogram calls, which would manifest as cycles in the program call graph. Naturally, this extension also applies to the core semantics in Fig. 2.4.

```
program sample;
  input in, n;
  local count, lastcount;
  output out;

  out       := if count=0 then in else Nil fi;
  count     := if lastcount=n-1 then 0 else lastcount+1 fi;
  lastcount := n-1 fby count;
endprogram;
```

Figure 2.6: `sample` definition

```
program fby-every;
  input in1, in2, n;
  local count, lastcount;
  output out;

  out       := if count=0 then laststate else Nil fi;
  state     := if count=0 then in2 else laststate fi;
  laststate := in1 fby state;
  count     := if lastcount=n-1 then 0 else lastcount+1 fi;
  lastcount := n-1 fby count;
endprogram;
```

Figure 2.7: `fby-every` definition

Constants			
$\mathtt{c}\,\mathtt{every}\,\mathtt{n}$	$\mapsto$	$\mathtt{sample}(\mathtt{c},\mathtt{n})$	for $\mathtt{c} \in C, \mathtt{n} \in \mathbb{N}$
Variables			
$\mathtt{x}\,\mathtt{every}\,\mathtt{n}$	$\mapsto$	$\mathtt{sample}(\mathtt{x},\mathtt{n})$	for $\mathtt{x} \in X, \mathtt{n} \in \mathbb{N}$
Operators			
$\mathtt{op}(\mathtt{e})\,\mathtt{every}\,\mathtt{n}$	$\mapsto$	$\mathtt{op}(\mathtt{e}\,\mathtt{every}\,\mathtt{n})$	for $\mathtt{e} \in E, \mathtt{n} \in \mathbb{N}$, $op \in \{+,-,*,/,=,<>,<,>,\mathtt{not},\mathtt{and},\mathtt{or},\mathtt{ite}\} \cup SP$
$\mathtt{e}_1\,\mathtt{fby}\,\mathtt{e}_2\,\mathtt{every}\,\mathtt{n}$	$\mapsto$	$\mathtt{fby\text{-}every}(\mathtt{e}_1,\mathtt{e}_2,\mathtt{n})$	for $\mathtt{n} \in \mathbb{N}$
Equations			
$(\mathtt{x} := \mathtt{e})\,\mathtt{every}\,\mathtt{n}$	$\mapsto$	$\mathtt{x} := \mathtt{e}\,\mathtt{every}\,\mathtt{n}$	for $\mathtt{n} \in \mathbb{N}$
Tupling			
$(\mathtt{e}_1,\mathtt{e}_2)\,\mathtt{every}\,\mathtt{n}$	$\mapsto$	$(\mathtt{e}_1\,\mathtt{every}\,\mathtt{n},\mathtt{e}_2\,\mathtt{every}\,\mathtt{n})$	for $\mathtt{n} \in \mathbb{N}$
$(\mathtt{eqn}_1;\mathtt{eqn}_2;)\,\mathtt{every}\,\mathtt{n}$	$\mapsto$	$\mathtt{eqn}_1\,\mathtt{every}\,\mathtt{n};\mathtt{eqn}_2\,\mathtt{every}\,\mathtt{n};$	for $\mathtt{n} \in \mathbb{N}$
Programs			
$(\mathtt{P}(\bar{\mathtt{x}});\mathtt{eqns};)\,\mathtt{every}\,\mathtt{n}$	$\mapsto$	$\mathtt{P}(\bar{\mathtt{x}});\mathtt{eqns}\,\mathtt{every}\,\mathtt{n}$	for $\mathtt{n} \in \mathbb{N}$

Figure 2.8: Translation rules for `every` operator

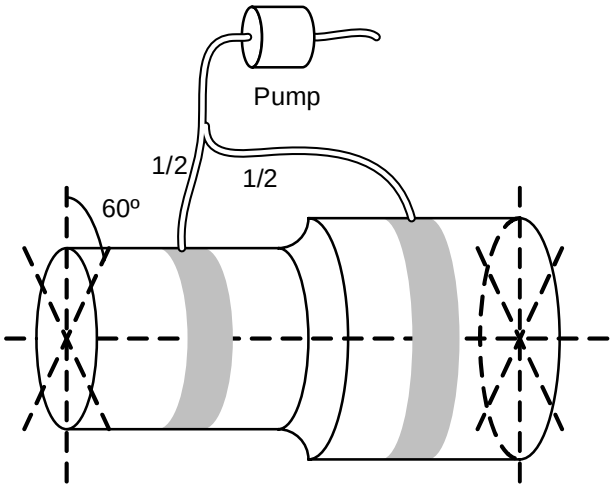


Figure 2.9: The oil pump illustrated

Example (Oil pump monitor). An oil pump is used to continuously lubricate a shaft rotating at high speed. The shaft is mounted with two bearings A and B of different diameter. The amount of oil fed by the pump must be closely monitored in order to avoid friction and eventual overheatage on the one hand, and overdosage on the other. A software-controlled monitor for the oil pump is to be built as follows (Fig. 2.9):

- An angular sensor provides a tick signal every $60°$ of shaft rotation.
- An oil flow sensor provides a sample of the oil flow rate through either one of the individual oil lines (`flow`) every $120°$ of shaft rotation. The unit of measurement is $\frac{\mu\mathrm{l}}{\mathrm{tick}}$.
- The perimeter of the shaft at the bearing A is $200\mathrm{mm}$, while the perimeter at the bearing B is $300\mathrm{mm}$. Both bearings have the same width, and the oil flow is split evenly between bearing A and bearing B.
- Experience shows that volume must be monitored every $100\mathrm{mm}$ of glide travel.
- The amount of oil dispensed should be calculated separately for the two bearings.

```
program oil-pump;
  input flow;
  output vol_a, vol_b;

  program integrator;
    input in, dt;
    var state;
    output out;

    out   := 0 fby state;
    state := in*dt + out;
  endprogram;

  program gain;
    input in;
    output out;

    out := in*3/2;
  endprogram;

  vol_a := gain(hold(vol_b)) every 3;
  vol_b := integrator(flow, 2) every 2;
endprogram;
```

Figure 2.10: Program `oil-pump`

The implementation of the volume measurement for the two bearings as a synchronous program is shown in Fig. 2.10 (we omit the declaration of the `hold` operator). Note that `vol_b` denotes the volume of oil dispensed for bearing B, and is calculated by integrating over `flow` with time constant 2. `vol_a` is the oil volume for bearing A, and is computed by sampling `vol_b`, and multiplying by $\frac{3}{2}$.

Partitioning normal forms. For the subsequent treatment in Chapters 3 and 4, we shall resort to a canonical structure for SSDL programs. Let `P` be an SSDL synchronous program with variable set X, where `P`'s equations are of the form (subprogram declarations for $\mathtt{P}_{(l+1)}, \ldots, \mathtt{P}_i, \ldots, \mathtt{P}_n$ are not shown):

$$\begin{array}{rcl} \bar{\mathtt{x}}_{O(l+1)} & := & \mathtt{P}_{l+1}\,(\bar{\mathtt{x}}_{I(l+1)}); \\ \vdots & \vdots & \vdots \\ \bar{\mathtt{x}}_{Oi} & := & \mathtt{P}_i \quad (\bar{\mathtt{x}}_{Ii}); \\ \vdots & \vdots & \vdots, \\ \bar{\mathtt{x}}_{On} & := & \mathtt{P}_n \quad (\bar{\mathtt{x}}_{In}); \end{array}$$

where for $1 \leq i \leq n$, the $\bar{\mathtt{x}}_{Oi}$ are nonempty *output variable* tuples such that each individual variable $\mathtt{x} \in X_L \cup X_O$ is a component of exactly one tuple, and $\bar{\mathtt{x}}_{Ii}$ are possibly empty *input variable* tuples, such that individual variables $\mathtt{x} \in X$ appear in the tuples arbitrarily often. Clearly, any valid SSDL program can be written in this form. We call it the *partitioning normal form* (PNF), as it canonically partitions the SSDL program into subprograms ready for distributed implementation.

A derived normal form is the *multiclock partitioning normal form* (MPNF), where an SSDL program's equations obey the following structure:

$$\begin{array}{rcccccl} \bar{\mathtt{x}}_{O(l+1)} & :=\big[\,\mathtt{e}_{l+1} & \mathtt{fby}\big] & \mathtt{P}_{l+1} & (\mathtt{hold}(\bar{\mathtt{x}}_{I(l+1)})) & \mathtt{every}\, n_{l+1}; \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \bar{\mathtt{x}}_{Oi} & :=\big[\,\mathtt{e}_i & \mathtt{fby}\big] & \mathtt{P}_i & (\mathtt{hold}(\bar{\mathtt{x}}_{Ii})) & \mathtt{every}\, n_i; \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \bar{\mathtt{x}}_{On} & :=\big[\,\mathtt{e}_n & \mathtt{fby}\big] & \mathtt{P}_n & (\mathtt{hold}(\bar{\mathtt{x}}_{In})) & \mathtt{every}\, n_n; \end{array}$$

Angled brackets $[\,]$ indicate that the delay operators `fby` surrounding subprogram references $\mathtt{P}_i$ are optional, the $\mathtt{e}_{l+1}, \ldots, \mathtt{e}_i, \ldots, \mathtt{e}_n$ denote expressions for initial values, and $\mathtt{hold}(\bar{\mathtt{x}}_{Ii})$ indicates that at least those individual variables of $\bar{\mathtt{x}}_{Ii}$ which are on a different clock than subprogram $\mathtt{P}_i$ are surrounded by a `hold` operator.

In effect, SSDL programs in MPNF combine subprograms in a similar way as PNF, but in addition, streams for communication between subprograms may be delayed, and subprograms may be computed at a slower rate through the `every` operator. We note that, for instance, `oil-pump` is in MPNF, with subprograms `gain` and `integrator` sampled by `every`.

2.2.4 Non-reactive programs and causality analysis

Reactive programs run indefinitely, and map infinite input sequences to infinite outputs. Having guaranteed solely the existence of a least fixed point, it is not necessarily clear that this fixed point is nonempty, and indeed yields infinite streams, therefore capturing the intuitive meaning of a reactive program. To illustrate this point, it is helpful to look at two examples (Fig. 2.2.4): Seen as a constraint over the progression of `x` in time,

```
program agree;
  output x;

  x := x;
endprogram;
```

```
program contradict;
  output x;

  x := not x;
endprogram;
```

Figure 2.11: Two non-reactive programs

program `agree` is inherently ambiguous: any finite and infinite sequence for `x`, including the empty sequence, would satisfy the constraint. In our operational (Kahn) interpretation, the fed back process for `x` would have to wait indefinitely for its own result, resulting in a deadlock, and yielding the empty stream. As another example, `contradict` seems to have no valid behaviors under both interpretations, except for the empty sequence. Least fixed point semantics, as in SSDL, maps `x` to the empty stream ε in both programs, therefore providing a unique semantics in both cases. However, such empty runs do not correspond to the intuitive notion of a reactive program, which continuously operates for an indefinite amount of time, and therefore yields infinite streams in the semantics for infinite inputs.

Consequently, both `agree` and `contradict` are *rejected* as SSDL programs, as they are not reactive. To be more precise, we will define the notion of a valid SSDL program in Def. 2.8. As an additional definition, we introduce variable dependency relations in Def. 2.7.

Definition 2.7 (Variable dependency relation $\rightsquigarrow^+$). *For a variable set X, let $\rightsquigarrow \subseteq X \times X$ be the reduced variable dependency relation defined as follows:*

`x` $\rightsquigarrow$ `y` *if* `x` *appears on the right-hand side of* `y`*'s defining equation, and at least one of* `x`*'s occurences on the right-hand-side is not in a delayed (second argument of* `fby`*) context.*

`x` $\not\rightsquigarrow$ `y` *otherwise.*

Then the variable dependency relation $\rightsquigarrow^+ \subseteq X \times X$ *is defined as the transitive closure of* $\rightsquigarrow$.

Definition 2.8 (Valid SSDL program). *An SSDL program is valid if its variable dependency relation* $\rightsquigarrow^+$ *is irreflexive, that is, for all* `x` *it holds that* `x` $\not\rightsquigarrow^+$ `x`.

In particular, this analysis ensures that non-reactive programs such as `agree` or `contradict` are rejected as invalid. In some cases, however, the causality analysis is overly conservative, and rejects reactive programs. For instance, the program `no-causal-cycle` in Fig. 2.12 is rejected: according to Defs. 2.7 and 2.8, `y` depends on `z` and vice versa, leading to rejection of `no-causal-cyle`. However, `no-causal-cycle` could be compiled into a perfectly operational reactive program, as it will be true that a given value of `x` chooses either the first or the second term of `if.then.else.fi` in both equations, so no actual cycle will occur at run-time. SSDL shares this conservative causality analysis with other synchronous dataflow languages such as AutoFOCUS, or LUSTRE. In practice, as shown by a number of case studies, this restriction has not been a problematic issue so far. It is interesting to note, however, that more controlflow-oriented synchronous languages such as ESTEREL [Ber00] or STATECHARTS [Har87] typically have more sophisticated causality analysis, and would allow programs similar to `no-causal-cycle`.

```
program no-causal-cycle;
  input x;
  output y, z;

  y := if x then z else 0;
  z := if x then 0 else y;
endprogram;
```

Figure 2.12: Program `no-causal-cycle`

Synchronous runs as partially ordered sets. The stream-based semantic characterization of SSDL programs given in Section 2.2.2 is "declarative" in the sense that it defines *what* (in terms of its visible I/O behavior) the

```
program y-first;
  input x;
  output y, z;

  y := x + 1;
  z := y + 1;
endprogram;
```

```
program yz-both;
  input x;
  output y, z;

  y := x + 1;
  z := x + 2;
endprogram;
```

Figure 2.13: Two programs with identical stream semantics and different operationalizations

```
//y-first
//implementation

int x, y, z;

void loop() {
  y = x + 1;
  z = y + 1;
}
```

```
//yz-both
//1st
//implementation
int x, y, z;

void loop() {
  y = x + 1;
  z = x + 2;
}
```

```
//yz-both
//2nd implementation
int x, y, z;

void loop() {
  z = x + 2;
  y = x + 1;
}
```

Figure 2.14: Three C implementations

program computes, but not exactly *how* (in terms of the causal relationship between individual variables) the program computes it.

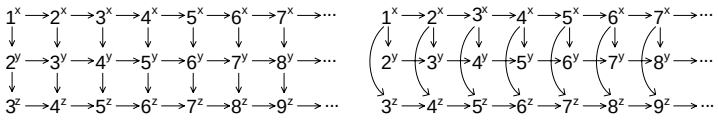
For instance, SSDL programs `y-first` and `yz-both` in Fig. 2.13 both are characterized by the identical denotational semantics, for $x \in V^\omega$, $y \in V^\omega$, $z \in V^\omega$:

$$y = x +^\omega 1^\omega, z = y +^\omega 1^\omega.$$

This stream-based characterization is sufficiently precise in the sense that `y-first` and `yz-both` cannot be distinguished by any context attached to the respective program by synchronous composition[6].

However, a straightforward sequential implementation, e. g. in terms of a C program as shown in Fig. 2.14, would yield different operational characteristics for both programs. Chapters 3 and 4 will be concerned with such operationalizations of synchronous programs. Consequently, the stream-based characterization alone will not be sufficient to reason about operationalizations. We need additional information about causality to know, for instance, whether `z` is computed based on `x`, or `y`, in programs `y-first` and `yz-both`.

[6]For a brief discussion of the related concept of full abstraction, see Chapter D in the appendix.



(a) Poset for a run of `y-first`

(b) Poset for a run of `yz-both`

The issue of declarative vs. operational semantics becomes clear if we, for a moment, consider *partially ordered sets* (posets) of symbol occurences instead of streams as a semantic characterization of synchronous dataflow programs. Each such poset characterizes one possible run for a program P. Two symbol occurences in the poset are ordered iff the occurence of one symbol causally precedes the occurence of the other. As an example, consider runs for both `y-first` and `yz-both`, where for the example runs, `x` is an increasing sequence of integers (Fig. 2.2.4).

We can easily show that this poset characterization indeed contains the "missing information" about causal orderings and operationalizations, yet also has all the inherent information from the stream semantics. To demonstrate this fact, both the stream semantics and operationalizations can be characterized in terms of the poset semantics as follows:

Streams as variable-wise projections of posets. For pairs of symbol occurences for *different* variables, remove any dependencies. The remaining poset is the union of totally ordered sets (or streams) of symbol occurences, one stream for each variable.

Operationalizations as linear extensions of posets. Consider an observer that observes a (potentially distributed) operationalization of an SSDL program such that any two symbol occurences are ordered in time, so observations are totally ordered, and the observation order respects causality. For a given run, *any linear extension* of the poset for the run is a possible operationalization of the run for this observer[7].

So a poset characterization of an SSDL program yields all the necessary information for a full semantic *and* operational characterization. Yet for reactive programs, such posets are infinite objects, and as a semantic characterization, they are even less "handy" and intuitive than streams.

[7]Note that this simple definition does not consider fairness and boundedness concerns. Chapter 4 will treat resource-bounded operationalizations of SSDL programs in more detail.

Posets are also not fully abstract for synchronous dataflow programs, as they capture more information than can be observed under the given compositional forms.

Variable dependency order. To capture causality, though, we note that the structure of inter-variable dependencies in valid SSDL programs is somewhat canonical: if a symbol occurence for variable `y` depends on a symbol occurence of another variable `x` in at least one step in at least one run, then there will be no run where `x` depends on `y`. We therefore turn to a finite *variable dependency order* over the variable set X, which relates variables directly instead of their symbol occurences. Two variables `x`, `y` depend on each other, written $\mathtt{x} \rightsquigarrow^* \mathtt{y}$, if for a given step, symbol occurences for `y` may be computed from `x` within the same step.

Definition 2.9 (Variable dependency order). *For a valid SSDL program with variable set X, the variable dependency order $\rightsquigarrow^* \subseteq X \times X$ is defined as the reflexive closure of $\rightsquigarrow^+$. For a variable $\mathtt{x} \in X$, we write $\rightsquigarrow^{*-1}(\mathtt{x}) = \{\mathtt{y} \in X \mid \mathtt{y} \rightsquigarrow^* \mathtt{x}\}$ for the inverse image of `x` through $\rightsquigarrow^*$.*

The variable dependency order is a partial order, as it is obviously reflexive and transitive, and also antisymmetric (see Lemma A.6 in the appendix). It is also qualitative in nature, as it only distinguishes between the presence and the absence of an immediate dependency between variables. For instance, the variable dependency order does not differentiate between an unrelated variable pair and a variable pair related by a delay (`fby`). As we will need this information, and also information on the number of delays between a pair, in Chapters 3 and 4, we define a *variable synchronization relation* next, which will carry the required information.

Definition 2.10 ((Reduced) variable synchronization relation $\xrightarrow{(.)}$). *For a valid SSDL program with variable set X, we define the reduced variable synchronization relation $\xrightarrow{(.)} \subseteq X \times \mathbb{N}_0 \times X$ as follows, for all $\mathtt{x}, \mathtt{y} \in X$, for all $n \in \mathbb{N}_0$:*

- $\mathtt{x} \xrightarrow{\mathtt{n}} \mathtt{y}$ *if* `x` *appears on the right-hand side of* `y`*'s defining equation, and for this equation, n is the least number of delay (*`fby`*) operators surrounding any reference (as second argument of* `fby`*) to* `x`*.*
- $\mathtt{x} \xrightarrow{\mathtt{n}\!\!\!\!\rightarrow} \mathtt{y}$ *otherwise.*

For instance, if `y := x + 0 fby x`, then $\mathtt{x} \xrightarrow{\mathtt{0}} \mathtt{y}$. Similar to taking the reflexive-transitive closure of $\rightsquigarrow$ for $\rightsquigarrow^*$, we'd like to construct a variable

synchronization relation $\xrightarrow{(.)}{}^{*} \subseteq X \times \mathbb{N}_0 \times X$, where successive weights for transitive pairs are added up, while for the case of multiple $(\mathtt{x}, \mathtt{y})$-dependencies, only the minimum weight is retained. We will need some lightweight definitions for constructing $\xrightarrow{(.)}{}^{*}$ as follows.

Definition 2.11 ((Uniquely) weighted ternary relation). *Let D be a set. A relation R is a weighted ternary relation over D if it is of the form $R \subseteq D \times \mathbb{N}_0 \times D$.*

A weighted ternary relation R is uniquely weighted if for each pair $d, d' \in D$, there exists at most one $n \in \mathbb{N}_0$ such that $(d, n, d') \in R$.

Definition 2.12 (Unweighting operator). *Let D be a set, and let R be a uniquely weighted ternary relation over D. We define an operator $Unweight : \wp(X \times \mathbb{N}_0 \times X) \to \wp(X \times X)$ as follows: for all $d, d' \in D$,*

$$\begin{array}{ll} (d, d') \in Unweight(R) & \textit{if } (d, 0, d') \in R \\ (d, d') \notin Unweight(R) & \textit{otherwise} \end{array}$$

Definition 2.13 (Functional form of weighted ternary relation). *Let R be a uniquely weighted ternary relation. We can write R in functional form as a function $R : (X \times X) \to \mathbb{N}_0 \cup \{\infty\}$, which is defined as follows: for all $d, d' \in D$,*

$$R(d, d') = \begin{cases} n & \textit{if } \exists n \in \mathbb{N}_0 \,.\, (d, n, d') \in R \\ \infty & \textit{otherwise} \end{cases}$$

Definition 2.14 (Minimum-weight reflexive-transitive closure). *Let R be a weighted ternary relation over set D. Based on R, we define an auxiliary relation $R^{\circledast} \subseteq D \times \mathbb{N}_0 \times D$ as the least relation such that*

$$\begin{array}{rll} & (d, 0, d) \in R^{\circledast} & \textit{for all } d \in D \\ (d, n, d') \in R \implies & (d, n, d') \in R^{\circledast} & \textit{for all } d, d' \in D, n \in \mathbb{N}_0 \\ \big((d, m, d') \in R^{\circledast} \wedge (d', n, d'') \in R^{\circledast}\big) \implies & (d, m{+}n, d'') \in R^{\circledast} & \textit{for all } d, d', d'' \in D, m, n \in \mathbb{N}_0 \end{array}$$

Then the minimum-weight reflexive-transitive closure of R, R^*, is defined as the least relation such that, for all $d, d' \in D$, for all $m \in \mathbb{N}_0$,

$$\big((d, m, d') \in R^{\circledast} \;\wedge\; m = \min\big\{n \;\big|\; (d, n, d') \in R^{\circledast}\big\}\big) \implies (d, m, d') \in R^*$$

where $\min\{n \mid (d, n, d') \in R^{\circledast}\}$ denotes the lower bound of the totally ordered set $\{n \mid (d, n, d') \in R^{\circledast}\} \subseteq \mathbb{N}_0$, and $\min \emptyset$ is assumed to be defined.

Clearly, $\xrightarrow{(\cdot)}$ is a weighted ternary relation over X. We denote as the *variable synchronization relation* $\xrightarrow{(\cdot)}^* \subseteq X \times \mathbb{N}_0 \times X$ the minimum-weight reflexive-transitive closure of $\xrightarrow{(\cdot)}$. For instance, if `z := 0 fby x + y` and `y := x`, then `x` $\xrightarrow{0}^*$ `z`. In the sequel, we shall use both the relational and the functional notation for $\xrightarrow{(\cdot)}^*$ interchangeably.

The dependency order provides additional information to the denotational semantics given by the stream relations above. We will use both $\rightsquigarrow^*$/ $\xrightarrow{(\cdot)}^*$ and the denotational semantics in order to define correct linearizations of a synchronous program. We note that $\rightsquigarrow^* = Unweight(\xrightarrow{(\cdot)}^*)$.

Summarizing the operational dependencies in trace posets as a simple dependency order over variables, $\rightsquigarrow^*$, is clearly only possible for language where this synoptic view is sound. For synchronous languages such as ESTEREL [Ber00] or various STATECHARTS [Har87] dialects, our approach based on a fixed and antisymmetric order $\rightsquigarrow^*$ would not suffice in the general case, as two variables may be mutually dependent. In a sense, this is one of the primary reasons why the discussion in this thesis centers around synchronous *dataflow* languages, which admit simple dependency orders over variables, as opposed to synchronous languages in general.

2.3 Related work

As already mentioned, there is a wealth of synchronous dataflow languages besides SSDL, all of which are much better suited for practical use. Prominent examples are LUSTRE and SIGNAL [BCE+03], and the discrete-time subset of Simulink [MW]. The AutoFOCUS tool and notation [HSE97] incorporates a synchronous notion of time, and its deterministic, recursion-free subset can be encoded using SSDL.

Related discussions of the synchronous paradigm and its suitability for engineering and programming have been given in [Kop92][BCGH94] [Ber00]. Of the cited works, [BCGH94] is most similar to this chapter as it focuses primarily on practical and theoretical aspects of dataflow synchronous languages, and explores some possibilities for semantic foundations. [THW01] discusses suitability issues and research directions for functional languages in the embedded and real-time sector.

Chapter 3

Two Implementation Schemes

This chapter describes two implementation schemes for synchronous dataflow programs. After a brief introduction to the considered platforms in Section 3.1, a singleprocessor implementation scheme is outlined in Section 3.2, where a synchronous dataflow program is partitioned into several threads running on one processor, and the composite multithreaded program implements the semantics of the synchrous dataflow program using inter-thread communication primitives. The other implentation scheme in Section 3.3 describes the multiprocessor case, where a synchronous dataflow program is split across several processors communicating over some event-triggered, bounded-jitter communication medium. Both implementation schemes are shown to preserve certain critical aspects of the dataflow program's semantics. In Section 3.4, finally, some related publications are described.

3.1 Platforms

Distributed embedded systems in the automotive field are typically implemented according to Fig. 3.1: A number of computing nodes or electronic control units (ECUs) perform computation in real-time, each using a real-time operating system to dispatch application processes. Communication between nodes is provided by one or several bus systems.

Operating Systems: OSEK/ERCOS^{EK}. OSEK [OSE01] is the dominating standard for automotive operating systems. As opposed to other operating systems from the embedded field, the OSEK standard is very restrictive with respect to dynamic, run-time features, while favoring static planning:

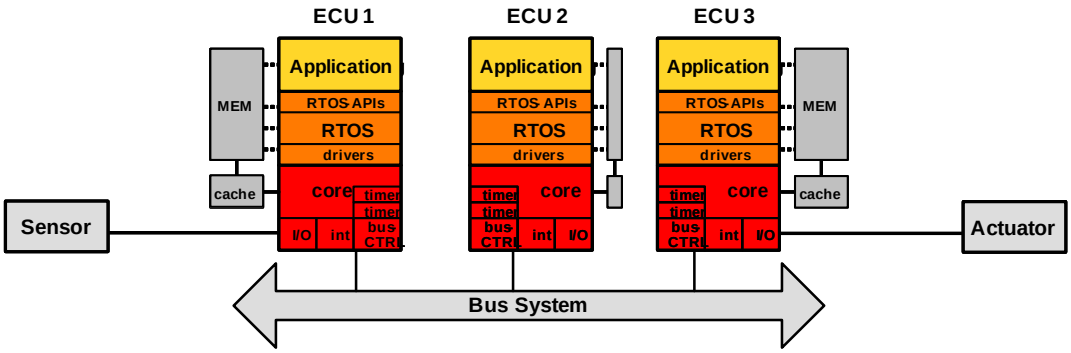


Figure 3.1: Automotive in-vehicle network (schematic)

Static memory allocation. There is no dynamically allocated memory heap in OSEK-based systems. Primitives such as `malloc` in C are not supported on the operating system level. Therefore, memory usage has to be planned at compile-time.

Statically assigned tasks. OSEK has no support for dynamically creating and destroying tasks. All tasks must be statically configured.

Restricted scheduling options. OSEK supports two kinds of task models, *basic tasks* and *extended tasks*. The latter model, in extension to the basic task model, supports a distinct activation mechanism for message-based communication. The scheduler in OSEK is a *fixed-priority* scheduler: each task is configured with a statically assigned priority. During run-time, when several tasks are ready to be released, the scheduler releases the task with the highest such priority. Other scheduling options are not available. The OSEK standard allows both preemptive and non-preemptive schedulers.

Limited primitives for inter-task communication. The OSEK standard suggests the usage of global variables for inter-task communication. Such variables may optionally be protected against data corruption by mutual-exclusion semaphores with priority ceiling protocol [SRL90] for inter-process communication (IPC). Our work relies on a different mechanism, which belongs to the class of *wait-free IPC* primitives, and closely resembles the state message mechanism provided by the ERCOS^{EK} [PMSB96] operating system for automotive applications.

Bus Systems: CAN. For automotive in-vehicle networks, we restrict our treatment to the popular CAN bus [Ets01], which is very widely used in

present-day vehicles. The multiprocessor scheme in Section 3.3 is suited for the event-triggered base protocols where communication may be statically bounded. CAN is a prominent example of this class of protocols.

CAN is based on the CSMA/CA principle of arbitration: connected hosts may concurrently try to access the medium after the current transmission has completed (Carrier Sense Multiple Access). After an initial arbitration phase, the medium is granted to the unique host with the highest priority. "Collisions" between different hosts, that is, two hosts sending a message at the same time, are avoided by an arbitration scheme which causes no temporal overhead for the arbitration itself (Collision Avoidance).

If application data is cleverly packaged into frames, the CAN bus can be used with a relatively small overall protocol overhead. This is partly due to the efficient arbitration mechanism, which uses an electrical resolution technique (dominant/recessive bits), and needs no additional handshaking for establishing a consensus. On the other hand, this same technique has the drawback that the maximum transmission rate is physically tied to wire length (maximum: 1 megabyte per second).

The basic protocol also includes an error correction mechanism, where corrupted messages are repeated until all receivers have acknowledged the receipt. This mechanism realizes an atomic broadcast for all but some subtle cases [RVA+98], making the protocol very well-suited for event-message delivery, and somewhat less suited for state-message delivery. We will elaborate on this issue in Section 3.3.

3.2 Singleprocessor implementation

As we have outlined in Chapter 1, the singleprocessor scheme is designed to realize the communication semantics of a synchronous program based on inter-task communication on a single processor. While the detailed schmeme will be given in Section 3.2.4, some preliminary remarks are in order. The particular problem of preservation via inter-task communication clearly does not arise if the program is realized as a single task: in fact, one may ask why multitasking is strictly necessary in the implementation of a synchronous program. As a motivation, one has to consider that, in embedded and real-time systems, the need for partitioning a program into several tasks with possibly multiple frequencies of periodic (or sporadic) activation may arise from the heterogeneity of a system's interface with its environment, and the various patterns of events and state samplings that the system has to react to. Consider the `oil-pump` example from Sec-

tion 2.2.3: Due to the different perimeters of the shaft at bearings A and B, and the uniform requirement of $100\mathrm{mm}$ part travel between samples, the sampling frequencies for the two bearings are different. Assigning both subprograms to a single task would, among other problems, lead to ineffiencies in processor utilization: The single task may be scheduled only to find out that, in the current step, there is nothing to do.

3.2.1 Subprograms, task partitions, and clocks

For the rest of this section, we shall assume that an SSDL program is in multiclock partitioning normal form (MPNF), as defined in Section 2.2.3. How does the notion of subprogram relate to task partitions? In our example, a natural mapping of `oil-pump` to a multitasking program would be to allocate one task for each subprogram: subprogram `integrator` is mapped to a task triggered every 2 ticks, while subprogram `gain` is mapped to a task triggered every 3 ticks.

Task partitions vs. multirate programs. We can generalize the strategy just applied to `oil-pump` to a generic singleprocessor implementation scheme as follows: Let `P` be an SSDL program in MPNF, and let $\mathtt{P}_1, \mathtt{P}_2, \ldots, \mathtt{P}_n$ be its immediate subprograms, which are composed with each other at potentially different clocks, as defined by the surrounding `every` statements. In MPNF, composition occurs in the common superprogram using equations, tuple constructors, the `hold` primitive, the `every` primitive, and possibly the `fby` operator. In the implementation scheme, the following rules apply:

- Each subprogram $\mathtt{P}_i$ is translated to one thread of sequential code.
- The code of each subprogram $\mathtt{P}_i$ is executed in a separate OS task T_i.

Note that, at first glance, this one-to-one mapping seems like a fairly restrictive, and possibly harmful, implementation pattern: a suitable subprogram (or component) partition on an abstract, conceptually oriented level may be very different from a suitable task partition on the implementation level. Subprograms, or architectural components, are frequently partitioned according to reuse concerns, coherency of interfaces, or organizational responsibility, to name a few criteria. A task partition, on the other hand, will typically be optimized with respect to performance, efficiency, and fault tolerance requirements, which often stand in conflict to the abovementioned aspects. For instance, in the AutoMoDe method

[BBR+05], the subprogram partition is incorporated in an architecture level called FDA (Functional Design Architecture), whereas a structure similar to the task partition is described by an architecture level called LA (Logical Architecture). To relate between FDA and LA, a number of restructuring steps are proposed for transitioning between the architecture levels. Fortunately, associativity and commutativity of composition in synchronous dataflow renders these restructuring steps comparatively easy. Using restructuring, a given program or group of subprograms can be regrouped to match a given task partition, so our simple one-to-one implementation pattern is applicable to much more general situations.

Clocks. Composition of subprograms with the `every` primitive assigns every subprogram a frequency of execution through the second parameter of `every` (clock). In SSDL, the notion of clock is restricted to periodic patterns in terms of the base tick, and the clock is simply an integer. The concept of a clock may also be applied more generically [BCE+03][BBR+05], but for the purpose of this thesis, the simple periodic `every` mechanism of SSDL shall suffice.

We can easily define a partial order over clocks as follows:

Definition 3.1 (Clock order.). *The clock order $\preceq$ is defined as a relation over $\mathbb{N} \times \mathbb{N}$ such that, for all clocks $n_1, n_2 \in \mathbb{N}$,*

$$n_1 \preceq n_2 \Leftrightarrow \exists k \in \mathbb{N}.k \cdot n_1 = n_2,$$

that is, $n_1 \preceq n_2$ iff n_1 is an integer divisor of n_2.

Based on $\preceq$ and equality $=$ over naturals $\mathbb{N}$, a strict order $\prec = (\preceq \backslash =)$ is defined.

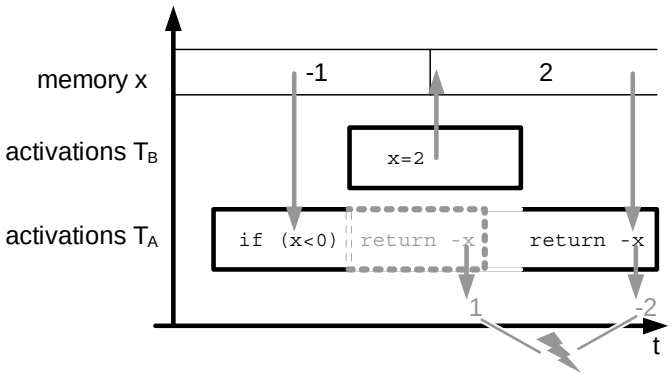
3.2.2 Preemptive scheduling and data consistency

We have seen in the introduction to this chapter that multitasking with fixed-priority schedulers is widely used in the automotive domain [OSE01]. In this section, our treatment will be furthermore restricted to *preemptive* schedulers. With the help of a preemptive scheduler, scheduling short-deadline and long-deadline tasks on the same processor is easier to solve, and under some more simplifying assumptions (which will be made explicit in the next paragraphs), one can apply the comparatively simple rate monotonic schedulability criterion [LL73].

```
extern int x;

int abs() {
  if (x < 0)
    return -x;
  return x;
}
```

(a) C implementation for `abs`



(b) Example timeline

Figure 3.2: Data inconsistency example

Data consistency. To illustrate the problem of safe data exchange, consider the example C implementation for an absolute value routine, `abs`, in Fig. 3.2(a). It returns the absolute variable of a global variable defined elsewhere, `x`. The routine `abs` first compares `x` to the value `0`, and then returns a value to the caller based on `x`. The strategy to use statically allocated global variables, instead of passing the argument as a function parameter, is very frequent in the embedded systems sector.

Now imagine `abs` being executed in a low priority task T_A, as illustrated in Fig. 3.2(b). Preemptive scheduling may lead to cases where the execution of the low priority task T_A is preempted by a higher priority task T_B. Under the assumption that the preempted task T_A reads value `x` and that the preempting task T_B writes the same memory location, inconsistencies may arise if the preemption occurs between two consecutive read operations at T_A, as shown in Fig. 3.2(b). The diagram shows what may happen if T_A gets interrupted during processing and no resource protection is implemented. Due to possible interference of T_B, the outcome of `abs` is "nondeterministic" in the sense that it depends on the precise runtime schedule. Furthermore, the illustrated behavior, where a negative value is returned to the caller, is generally not intended by the developer of `abs` and surrounding algorithms, so we allow ourselves to classify the data exchange as "unsafe" (may deviate from expected behavior).

To achieve safe exchange of data in a real-time system with preemptive scheduling, such as the system in Fig. 3.2, it has to be guaranteed that different tasks either do not directly access the same shared memory area, or

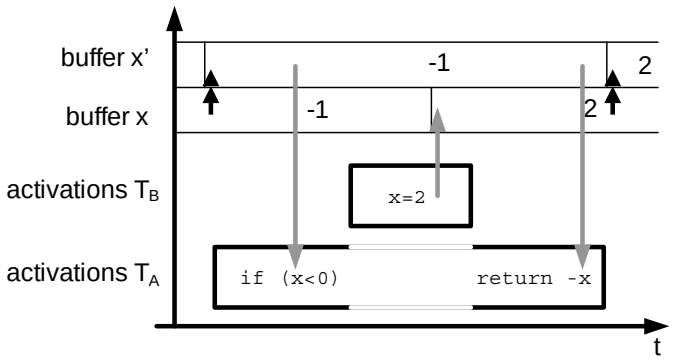


Figure 3.3: `abs` example with wait-free IPC

do so in a controlled way. In the latter case, safe data exchange may either be ensured by some suitable inter-process communication (IPC) mechanism, or it may be concluded that shared-memory exchange of data is safe, based on a static assertion that the possible run-time schedulings will never yield a hazardous situation. Generally speaking, a suitable IPC mechanism for real-time systems separates the memory space of tasks if required, performs the necessary run-time copy operations between memory areas if required, and incorporates suitable control and synchronization mechanisms.

To avoid unsafe data exchange, we end up with a desirable property called *data consistency*: For the duration between start and termination of a task T_A it should be guaranteed that all data locations which are accessed by T_A may change their value if and only if they are changed by T_A.

Wait-free IPC, and its realization by a double or triple buffer, is a suitable mechanism to achieve data consistency [Cla89][PMSB96][HPS02]. Fig. 3.3 illustrates the `abs` example with double buffering. The communication between tasks T_A (reader) and T_B (writer) now uses a double buffer, where T_B writes to one location, `x`, and T_A reads from another, `x'`. The underlying operating system or middleware is responsible for copying the contents from one location to the other prior to T_A being run. So possibly before the next invocation of T_A, `x` would be copied to `x'` by the operating system, indicated by the black double arrows in Fig. 3.3.

Note that the precise timing of this copy operation may be significant for a semantics-preserving translation of synchronous dataflow programs. We shall later see that our implementation scheme relies on a *time-* or *release-triggered* wait-free IPC mechanism. For the `abs` example, in the release-triggered variant, the copy operation is performed when the task is released, that is, as soon as the task is put in the scheduler's ready

queue. This mechanism may currently not be available in commercial operating systems or middlewares: To our knowledge, the closest to such a mechanism in a commercially available OS for the automotive market is the *run-triggered* wait-free IPC implemented in ERCOS^{EK}[PMSB96], where the copy operation is performed as soon as the task is actually run by the scheduler.

Scheduling and timing constraints. We now make explicit the assumptions about scheduling and timing constraints that will eventually translate, on the abstract level, to delay constraints imposed on the synchronous program. It is shown how clock information in the clocked subprograms correspond to actual real-time constraints for the implementation.

Fixed-priority, preemptive scheduling. We assume that the operating systems provides a fixed-priority preemptive scheduler, where task priorities are statically assigned. This corresponds to the OSEK standard [OSE01] for automotive operating systems.

Write suppression. The necessity for write suppression may arise in cases where "faster" tasks write to "slower" reader tasks, and the reader task needs consistent data over the period of its activation. The writer task is said to be *write-suppressing* with respect to the reader task if it will perform at most one write operation throughout any activation of the reader task. A formal definition of this criterion shall follow as part of the formal analysis in Section 3.2.6. We note that write suppression on the level of tasks corresponds to the inherent length-preservation on the level of SSDL programs: in the multiclock partitioning normal form of an SSDL program, a writer subprogram is always wrapped with `every` and `hold` primitives such that the input stream as seen by the reader subprogram is at the reader's clock. This is a natural consequence of the length-preservation criterion applied to the reader subprogram. So a semantics-preserving translation including the `every` and `hold`-induced sampling in the code of the writer will ensure write suppression. To this end, Section 3.2.3 will give a brief overview of code synthesis aspects.

Rate monotonicity. Within the fixed-priority scheduling framework, rate monotonicity (RMS) simply asserts that tasks with smaller periods are assigned higher priorities than tasks with greater periods [LL73].

Periodicity. Subprogram composition with `every` yields *periodic* subprograms: any subprogram is executed in a integer-valued, constant clock

with respect to the base tick.

A frequent additional assumption, which is not strictly necessary in our implementation scheme, is *time-periodicity*: Any subprogram is executed in a integer-valued, constant clock with respect to the base tick, *and* the base tick corresponds to some constant physical time period in the implementation. For instance, the `oil-pump` example from Section 2.2.3 is certainly periodic. Forcing time-periodicity in addition would correspond to the case of a shaft rotating at a constant rate. The time-periodicity constraint allows some further analysis from existing schedulability theory, where all times and periods are typically taken with respect to physical time.

Analysis for time-periodic systems. For any SSDL program in MPNF with subprograms $\mathtt{P}_1, \mathtt{P}_2, \ldots, \mathtt{P}_\mathtt{m}$ and subprogram clocks $n_1, n_2, \ldots, n_m$, each SSDL subprogram $\mathtt{P}_\mathtt{i}$ is associated with a task T_i. Each task T, in turn, is associated with a tuple (Per, WC, Rel, D) for period Per, worst case execution time WC[1], release times Rel, where $Rel : \mathbb{N} \to \mathcal{T}$ maps an activation index to the time domain $\mathcal{T}$, and deadline D. The processor utilization may be calculated as

$$U = \sum_{i=1}^{n} \frac{WC_i}{Per_i}$$

Then each valid schedule must meet the following constraints:

1. $Per_i = Per_0 \cdot c_i$, where c_i is clock of subprogram $\mathtt{P}_\mathtt{i}$, that is, the number of logical ticks that elapse between subsequent activations of $\mathtt{P}_\mathtt{i}$. Task periods are multiples of the *base period* Per_0 of the system. Per_0 must be specified by the developer. Each period corresponds to a base tick in the synchronous program.

2. $Rel_i(j) = j \cdot Per_i$. Tasks are released at the beginning of their period.

3. $D_i = Per_i$. The deadline of each task is equal to its period.

4. $U \leq m \cdot (2^{\frac{1}{m}} - 1)$. For m tasks mapped to a processor, keeping the processor utilization equal to or below the given (statically computable) upper bound ensures RMS schedulability [LL73]. Note that for task sets with harmonic frequencies (all subprogram clocks are related), higher upper bounds for utilization may be derived.

[1] Compile-time determination of this quantity leads to the problem of worst case execution time (WCET) analysis [KP02], which is not covered here.

5. $WC_i \leq D_i$. The worst case execution time is less than or equal to the deadline. This constraint follows indirectly from 3. and 4.

3.2.3 Sequential code synthesis

This section shall briefly summarize the synthesis of sequential code from synchronous dataflow programs. This topic has been extensively researched in the literature [HRR91][ABG95], and efficient code generators are firmly established in the commercial market [dSP][MW].

We have seen in Chapter 2 that variables in synchronous dataflow languages are partially ordered by a variable dependency order $\rightsquigarrow^*$, as defined in Def. 2.9. The partial order reflects possible schedulings of variable assignments within one step of execution. Clearly, the straightforward approach is to translate the equations of the synchronous program to an equivalent sequence of assignment operations in a sequential language. However, the total order of such sequential assignments will matter, as `a:=b; b:=c;` is generally not equivalent to `b:=c; a:=b;` in sequential languages. We end up with a total order of assignments generated from the partial variable order in the synchronous program.

It is easy to show that such an order of assignments in the code is correct if it corresponds to a linear extension of the variable dependency order $\rightsquigarrow^*$. Optimizations may prefer a certain total order: certain schedulings may use less assignment statements and potentially less additional memory locations than others. We shall illustrate the sequential code synthesis process with a small example.

Example (Differentiator). Consider the example of the SSDL program `diff` for a discrete differentiator, and a more modular variant, `modular-diff`, both shown in Fig. 3.4. In the case of `modular-diff`, we shall concentrate on subprogram `dt-delay`.

For `diff` and `dt-delay`, respectively, the variable dependency order, $\rightsquigarrow^*$, is shown in Fig. 3.5.

The solid edges correspond to pairs in the reflexive-transitive reduction of $\rightsquigarrow^*$. Dashed edges indicate additional desirable scheduling dependencies, which are introduced from variable $\mathtt{y} \in X$ to variable $\mathtt{x} \in X$ exactly if the variable synchronization relation yields $\mathtt{x} \xrightarrow{1}{}^{*} \mathtt{y}$. We assume that for any pair $\mathtt{x}, \mathtt{y} \in X$, if $\mathtt{x} \xrightarrow{\geq 2}{}^{*} \mathtt{y}$, then there exists $\mathtt{z} \in X$ such that $\mathtt{x} \xrightarrow{1}{}^{*} \mathtt{z}$ and $\mathtt{z} \xrightarrow{\geq 1}{}^{*} \mathtt{y}$. If this is not the case, it is always possible to obtain a program conforming with the constraint by introducing additional variables.

```
program diff;
  input in, dt;
  var local;
  output out;

  local := in/dt;
  out   := local
           - (0 fby local);
endprogram;
```

```
program modular-diff;
  input in, dt;
  output out;

  program dt-delay;
    input in, dt;
    output imm, del;

    imm := in/dt;
    del := 0 fby imm;
  endprogram;

  (imm, del) := delay(in, dt);
  out        := imm - del;
endprogram;
```

Figure 3.4: Programs `diff`, `modular-diff`

Figure 3.5: Variable dependency order $\rightsquigarrow^*$ for `differentiator`, `dt-delay`

```
//information passing
int del, imm;
//state
int imm_1 = 0;

void dt_delay(
    int in, int dt) {
  imm   = in/dt;
  del   = imm_1;
  imm_1 = imm;
}
```

```
//information passing
int del;
//information passing + state
int imm = 0;

void dt_delay(
    int in, int dt) {
  del = imm;
  imm = in/dt;
}
```

Figure 3.6: Two C implementations of `dt-delay`

```
//information passing
int out, local;
//state
int local_1 = 0;

void diff(int in, int dt) {
  local   = in/dt;
  out     = local - local_1;
  local_1 = local; //copy
endprogram;
```

Figure 3.7: C implementation of `diff`

If the additional desirable scheduling dependency can be obeyed, less assignment statements have to be executed, and possibly less memory usage occurs. In the `diff` example, because variable `imm` appears in a `fby` context as second parameter in the definition of `del`, it holds that $\texttt{imm} \xrightarrow{1}{}^{*} \texttt{del}$ according to Def. 2.10. Consequently, there is a dashed edge from `del` to `imm`, and so computing `del` before `imm` in a given step is desirable because it saves an assignment statement, and possibly an additional memory location. For `dt-delay`, the two possible assignment orders are

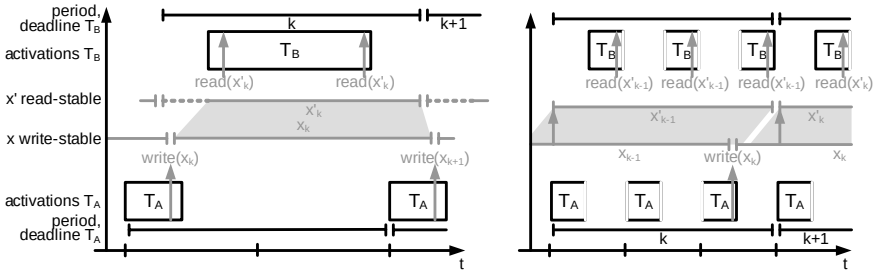
$$(\texttt{in}, \texttt{dt}) \rightarrow \texttt{imm} \rightarrow \texttt{del} \quad \text{and} \quad (\texttt{in}, \texttt{dt}) \rightarrow \texttt{del} \rightarrow \texttt{imm},$$

respectively.

The two corresponding C implementations are shown in Fig. 3.6. Note that the left-hand side implementation needs an additional variable declaration, `imm_1`, and an additional assignment statement. For `diff`, the only possible assignment order is

$$(\texttt{in}, \texttt{dt}) \rightarrow \texttt{local} \rightarrow \texttt{out},$$

and the dashed edge from `del` to `imm` cannot be obeyed. The chosen assignment order corresponds to the C implementation shown in Fig. 3.7.



(a) "A" alignment (b) "N" alignment

Figure 3.8: "A" and "N" alignments for two tasks

3.2.4 Three configuration rules for semantics-preserving inter-task communication

In this section, we finally state the implementation scheme for semantics-preserving inter-task communication. The scheme is based on an explicit case distinction with three different cases for relations of sender and receiver task clocks. In combination with a rate-monotonic task priority assignment, we shall find out that the use of double buffers is only necessary in two of three cases, while the remaining case can be covered by regular shared-variable interaction.

To get an intuitive understanding of the possible run-time activations, read instants, and write instants of writer and reader tasks, we shall first look at different classes of such scenarios, and obtain an intuitive grasp on the consequences for data consistency. According to the characterization of data consistency in Section 3.2.2, the use of double or triple buffers is generally dictated if no static assertion about the safety of simple shared-memory exchange of data can be obtained under the given scheduling assumptions. We shall therefore differentiate between a general situation where the safety assertion holds without additional buffering, henceforth called *"A" alignment*, and a situation where shared-memory exchange alone is not sufficient for safe data exchange, called *"N" alignment*.

"A" and "N" alignments. Consider a task T_A writing a variable `x`, and a task T_B reading the same variable under the local name of `x'`. Both tasks are assumed to have the same period. Figs. 3.8(a) and 3.8(b) are schematic depictions of the inter-task communication timing:

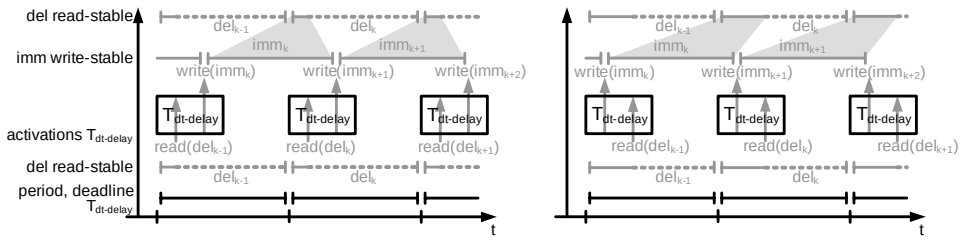
"A" alignment. In Fig. 3.8(a), the value written by T_A, `x`, is not changed

(stable) between write events: these events are indicated by grey "write" arrows. We indicate these intervals by the "x write-stable" bar. Similarly, for data consistency, T_B will require a stable value of `x'` as indicated by the "`x'` read-stable" bar. According to our definition of data consistency in Section 3.2.2, "`x'` read-stable" must be aligned with T_B's period: the dashed part of the bar indicates intervals where it can be statically asserted that T_B will never be activated, hence read-stability of `x'` is not actually needed.

We call this first kind of alignment between write-stable and read-stable intervals "A" because of the A-like shape of the light grey polygon relating the intervals. Essentially, the "`x'` read-stable" intervals are subsumed in their corresponding "x write-stable" intervals, so "A"-aligned communication can be performed with a single shared variable, and does not require extra memory for communication. In the example, T_A may write to the shared variable that is read by T_B. Formally, "A" alignment corresponds to the constraint that, for all possible instants, the current read event for T_B is after the current write event for T_A, and the read event for T_B precedes the next write event for T_A.

"N" alignment. In Fig. 3.8(b), the subscript k indicates equal values: $\mathtt{x}_k$ and $\mathtt{x'}_k$ reference the same value. This time, the light grey polygon relating the write-stable and read-stable intervals has a slanted shape: the interval for "x write-stable$_k$" and "`x'` read-stable$_{k-1}$" overlap, therefore a double buffer is required. Because the polygon shape resembles a slanted N, we call it the "N" alignment. Formally, "N" alignment corresponds to the constraint that, for all possible instants, the current read event for T_B is after the current write event for T_A, and the read event for T_B precedes the write event *after* the next write event for T_A. So it is both graphically and formally clear that "N" is a generalization of "A".

We can also easily relate the "A" and "N" alignments to the communication of a task with itself for implementing delays, as outlined in Section 3.2.3. Fig. 3.9 depicts timelines showing the write-stable intervals for variable `imm` and the read-stable intervals for variable `del` for the two implementation variants for `dt-delay` according to Fig. 3.6. While assignment order $(\mathtt{in}, \mathtt{dt}) \rightarrow \mathtt{del} \rightarrow \mathtt{imm}$ yields the "A" alignment shown in Fig. 3.9(a), assignment order $(\mathtt{in}, \mathtt{dt}) \rightarrow \mathtt{imm} \rightarrow \mathtt{del}$ corresponds to the "N" alignment depicted in Fig. 3.9(b). In the second case, the "N" alignment mandates additional copying and buffering. If a task is exclusively communicating with itself, the copying and buffering can be achieved by the introduction



(a) "A" alignment, no extra state variable for `imm`

(b) "N" alignment, extra state variable for `imm`

Figure 3.9: "A" and "N" alignments for one task

of the additional variable `imm'`, and by an additional assignment statement, as described in Section 3.2.3.

Having explored different alignments of run-time activations, read instants, and write instants of sender and receiver tasks, we are now ready to define three configuration rules listed in Fig. 3.10, each one suited for a different relation of sender and receiver task clocks. The three rules refer to pair-wise configurations of tasks, where T_i is the writer task and T_j is the reader task of the pair. c_i are c_j the respective clocks, which are determined by the `every` statements surrounding the subprogram references corresponding to T_i and T_j.

The first rule concerns the case $c_i \prec c_j$: We observe that, under the rate-monotonic priority assignment, the higher activation rate of T_i incurs that T_i has a higher priority than T_j. Consequently, the writer task T_i will never be preempted by the reader task T_j. Combined with the write-suppression criterion for communication from T_i to T_j, this ensures data consistency for the receiver task T_j according to "A" alignment by default, so a single shared variable is sufficient for safe communication. For a second rule, we consider the case $c_i \succeq c_j$: In this case, the writer task T_i may have lower priority than reader T_j and thus may be preempted by T_j, but because clocks c_i and c_j are harmonic, any instant of termination for T_i's period coincides with some instant of termination of one of T_j's periods. Communication can thus be understood as a single "N" alignment in reader-writer communication, and usage of a double buffer is sufficient for safe communication. The remaining case, where c_i and c_j are not related via $\preceq$, warrants the use of a triple buffer. The triple buffer effectively realizes a sequential arrangement of two "N"-aligned copies. Each of the three cases will be illustrated with an example in the following section.

The following rules yield the minimum necessary number of communication buffers for semantics-preserving inter-task communication, depending on the clock relationship between any two communicating tasks T_i (writer), T_j (reader):

$c_i \prec c_j$**.** Undelayed communication. Shared-variable communication ("A" alignment). See Section 3.2.5 for an example, see Theorems 3.6 and 3.7 for a formal treatment.

$c_i \succeq c_j$**.** Delayed communication. Double-buffer communication ("N" alignment), see Section 3.2.5 for an example, see Theorems 3.8 and 3.9 for a formal treatment.

$c_i \not\preceq\not\succeq c_j$**.** Triple-buffer communication (double "N" alignment), see Section 3.2.5 for an example, see Theorems 3.10 and 3.11 for a formal treatment.

Figure 3.10: Three configuration rules for semantics-preserving inter-task communication

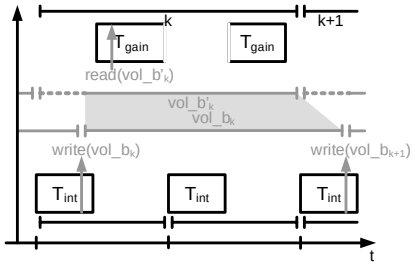
3.2.5 Examples

Example sequence for $c_i \prec c_j$ Fig. 3.11(a) shows an example of the `oil-pump` program introduced in Section 2.2, but with modified clocks for subprograms `integrator` and `gain`, respectively, to match the $c_i \prec c_j$ assumption. In the example, the clock of `integrator` is 2, and the clock of `gain` is 4, so clearly $c_{\texttt{int}} \prec c_{\texttt{gain}}$. Communication between subprograms `integrator` and `gain` through variable `vol_b` is immediate, so the result of `integrator` is processed by `gain` without an intermediary `fby` operator.

Naturally, `integrator` corresponds to a task $T_{\texttt{int}}$, `gain` corresponds to a task $T_{\texttt{gain}}$, and $T_{\texttt{int}}$'s period is half of $T_{\texttt{gain}}$'s period, consistent with the clocks in logical time. $T_{\texttt{int}}$ and $T_{\texttt{gain}}$ are released periodically at the beginning of their respective cycles. Both tasks run on the same processor, and are scheduled according to the rate monotonic policy, so $T_{\texttt{int}}$ has higher priority than $T_{\texttt{gain}}$. For instance, in the example timeline shown in Fig. 3.11(b), $T_{\texttt{gain}}$ is preempted by $T_{\texttt{int}}$ at the latter's second release. Note that periods and deadlines of $T_{\texttt{int}}$ and $T_{\texttt{gain}}$ are indicated by the black bars in Fig. 3.11(b).

Now because of $T_{\texttt{int}}$'s higher priority, `vol_b` will never actually be read before $T_{\texttt{int}}$ has finished its computation, and `vol_b` has been written. We

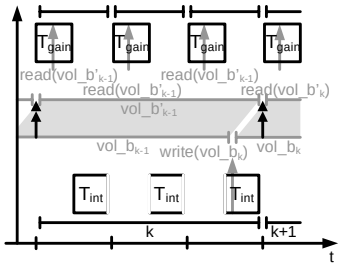
```
vol_a:=gain(hold(vol_b)) every 4;
vol_b:=integrator(flow, 2) every 2;
```



(a) `oil-pump` composition (b) Timeline

Figure 3.11: Example sequence for $c_i \prec c_j$

```
vol_a:=gain(hold(vol_b)) every 1;
vol_b:=0 fby integrator(flow, 3) every 3;
```



(a) `oil-pump` composition (b) Timeline

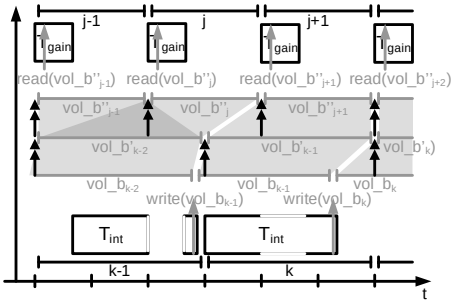
Figure 3.12: Example sequence for $c_i \succeq c_j$

indicate this by a dashed bar for "`vol_b` read-stable" during $T_{\texttt{int}}$'s activation. Therefore, we can safely use one shared variable for both `vol_b` and `vol_b'` . Because the written variable and the read variable correspond to the same memory location, no double buffering is needed, and the operating system does not have to perform an explicit copy operation. The example therefore illustrates that communication from fast to slow processes does not require introduction of a delay in the model, and that the case of related clocks with $c_i \prec c_j$ can be realized with shared-variable communication.

Example sequence for $c_i \succeq c_j$. Fig. 3.12(a) depicts the same `oil-pump` program from Section 2.2, again with modified clocks in order to match the $c_i \succeq c_j$ assumption. `integrator`'s clock is 3, and `gain`'s clock is 1. In the composition of A and B, a delay is imposed, corresponding to the delay operator in SSDL used for obtaining `vol_b` from the result of `integrator`.

Fig. 3.12(b) shows how the delay relates to the timeline of two associated tasks $T_{\texttt{int}}$ and $T_{\texttt{gain}}$. Because $T_{\texttt{int}}$ is slower than $T_{\texttt{gain}}$, and the clocks

```
vol_a:=gain(hold(vol_b)) every 3;
vol_b:=0 fby integrator(flow, 2) every 2;
```



(a) Composition in SSDL (b) Timeline

Figure 3.13: Example sequence for $c_i \not\perp\!\!\!\not\perp c_j$

are harmonic, the "vol_b' read-stable" period can be safely extended to $T_{\texttt{int}}$'s period.

If all tasks meet their deadlines, for the k-th activation of $T_{\texttt{gain}}$, $T_{\texttt{gain}}$ will never read vol_b'$_k$ before the corresponding value vol_b has been written for index k. We can therefore safely associate each read value vol_b'$_k$ with a corresponding written value vol_b$_{k-1}$ for valid index k. The difference in indices corresponds to a delay (fby) in the oil-pump program. The black double-headed arrows indicate buffer copy operations or buffer switches which have to be performed by the operating system.

Example sequence for $c_i \not\perp\!\!\!\not\perp c_j$. Fig. 3.13(a) shows our third example, the original oil-pump from Section 2.2, where the two subprograms have unrelated clocks, $c_i \not\perp\!\!\!\not\perp c_j$. $T_{\texttt{int}}$ and $T_{\texttt{gain}}$ are the two corresponding tasks. Rate monotonicity implies that $T_{\texttt{int}}$'s priority is higher than $T_{\texttt{gain}}$'s.

Note that the "vol_b read-stable" intervals are now completely unaligned with the "vol_b write-stable" intervals. This warrants the use of a *triple* buffer, involving three copies vol_b, vol_b', and vol_b". vol_b is the buffer written by integrator, vol_b' is a delayed, "safe" copy of vol_b which is aligned with $T_{\texttt{int}}$'s period, and vol_b" is sampled from vol_b' aligned with $T_{\texttt{gain}}$'s period.

Again, the black double-headed arrows indicate buffer copy operations or buffer switches performed by the operating system. Note that copy operations are assumed to take very little time, and are regarded as atomic. If two dependent copy operations are scheduled for the same instant, the causal order must match the data flow. In the example, this is the case for the $j-1$-th and $j+2$-th activation of $T_{\texttt{int}}$, where the vol_b→vol_b' copy must be performed before the vol_b'→vol_b" copy.

Beyond the exemplary treatment of the current section, we can formally demonstrate that for each of the three cases, the real-time semantics of communication and synchronization coincides with the ideal semantics inherent in the implemented SSDL program. The formal analysis illustrating this coincidence shall be the subject of the next section.

3.2.6 Formal analysis

This section summarizes the formalization and proofs for semantics-preserving communication with wait-free IPC. The formalization will focus on the establishment of the following two types of properties:

Preservation: Does the communication in the implementation reflect the communication in the synchronous program? For instance, the logical-time indices of messages read by a reader must coincide in the ideal semantics and in the implementation semantics.

Timing consistency: Does the possible temporal interleaving of `read`s and `write`s actually agree with the assumptions made for the preservation proof?

We shall further motivate the need for showing timing consistency in addition to preservation after introducing the notion of *time domains*, *local indices*, and *clock functions* in the next paragraphs, and after extending our simple formalization of tasks from Section 3.2.2 to a richer set of time functions.

Time domains, local indices, clock functions. As defined in Section 2.2.2, an SSDL subprogram can essentially be understood as a function mapping inputs to outputs. Equivalent to using a global index, and using `every` and `hold` for composition based on `Nil`-interleaved streams, we can map from a time domain common to all subprograms, $\mathcal{T}$, to local indices, $\mathbb{N}$, using monotonic *clock functions* C. In this alternative framework, streams are non-`Nil`-interleaved sequences which are interpreted with respect to their local indices. The common time domain must be chosen so that it temporally resolves all possible events, and possibly more. Monotonic *time functions* are the dual of clock functions in this framework, mapping from local indices to an instant in the global time domain.

We demonstrate the use of time domains, local logical clocks, and clock functions with an example. For the example configuration of `oil-pump` shown in Fig. 3.13, let us assume a time-periodic implementation with

time domain $\mathcal{T} = \mathbb{R}^+ = \{t \in \mathbb{R} \mid t \geq 0\}$ and base period Per_0. For demonstration, let Per_0 be 10.0, where the physical time unit associated with $\mathcal{T}$ is milliseconds. Individual periods $Per_{\texttt{int}}, Per_{\texttt{gain}}$ are defined according to the time-periodicity definition in Section 3.2.2: $Per_{\texttt{int}} = c_{\texttt{int}} \cdot Per_0 = 20.0$ and $Per_{\texttt{gain}} = c_{\texttt{gain}} \cdot Per_0 = 30.0$. Then clock functions $C : \mathcal{T} \to \mathbb{N}$, where C denotes one of $\{C_{\texttt{int}}, C_{\texttt{gain}}\}$, and release time functions $Rel : \mathbb{N} \to \mathcal{T}$, where Rel denotes one of $\{Rel_{\texttt{int}}, Rel_{\texttt{gain}}\}$, can be defined as follows:

$$\begin{aligned} C(t) &= \left\lfloor \frac{t}{Per} \right\rfloor + 1 \\ Rel(n) &= (n-1) \cdot Per \end{aligned}$$

For composition, `hold` is naturally emulated by the fact that local logical clocks are used instead of `Nil`-interleaved sequences, and C always maps to last local index where the held value is available. For the example configuration of `oil-pump` in Fig. 3.13, we define the ideal semantics based on clock functions C and release time functions Rel as follows: For V the value set of an SSDL program, let $\mathit{flow}, vol_a, vol_b : \mathbb{N} \to V$ be the corresponding non-`Nil`-interleaved sequences on the semantics level, each defined for its local logical index set $n \in \mathbb{N}$. Let $[\![\texttt{gain}]\!] : (\mathbb{N} \to V) \to (\mathbb{N} \to V)$ and $[\![\texttt{int}]\!] : (\mathbb{N} \to V) \to (\mathbb{N} \to V)$ denote the semantics of `gain` and `integrator` over finite streams $V^* = \mathbb{N} \to V$, as defined in Section 2.2.2. Let $\big(Id + (.)\big) : \mathbb{Z} \to \mathbb{N} \to \mathbb{N}$, denote the function defined by, for all $k \in \mathbb{Z}$, $n \in \mathbb{N}$,

$$(Id+k)(n) = \begin{cases} n+k & \text{if } n+k > 0 \\ 0 & \text{otherwise} \end{cases} .$$

$(Id+(.))$ can be overloaded in an equivalent way to the time domain $\mathcal{T}$ in place of naturals $\mathbb{N}$. We write Id for $(Id+0)$, the identity function. Furthermore, we denote function composition with the $\circ$ operator, where for functions F, G over elements d, $F \circ G(d) = F(G(d))$. For instance, for a finite stream $x : \mathbb{N} \to V$, $x \circ (Id-1)$ mirrors the expression `c fby x` in the semantics[2]. `oil-pump`'s semantics would then be expressed as, for all $n \in \mathbb{N}$,

$$\begin{aligned} vol_a(n) &= \big([\![\texttt{gain}]\!](vol_b \circ C_{\texttt{int}} \circ Rel_{\texttt{gain}})\big)(n) \\ vol_b(n) &= \big([\![\texttt{int}]\!](\mathit{flow} \circ (Id-1))\big)(n) \end{aligned}$$

[2]To simplify the formalization, we assume that x_1 is equal to the initial value of the first parameter of the `fby` statement. We ensure that this simplification does not incur a loss of generality for the correctness proof.

In the definition of vol_b, chaining with $(Id-1)$ realizes the unit delay introduced by the `fby` operator. An example sequence for flow and vol_b is shown in Fig. 3.14.

$n_{\texttt{int}}$	1	2	3	4	5	...
$\mathit{flow}(n_{\texttt{int}})$	0	1	2	1	1	...
$(\mathit{flow} \circ (Id-1))(n_{\texttt{int}})$	0	0	1	2	1	...
$\big([\![\texttt{int}]\!](\mathit{flow} \circ (Id-1))\big)(n_{\texttt{int}})$	0	0	0	1	3	...
$vol_b(n_{\texttt{int}})$	0	0	0	1	3	...
$n_{\texttt{gain}}$	1	2	3	4	...	
$\mathit{Rel}_{\texttt{gain}}(n_{\texttt{gain}})$	0.0	30.0	60.0	90.0	...	
$(C_{\texttt{int}} \circ \mathit{Rel}_{\texttt{gain}})(n_{\texttt{gain}})$	1	2	4	5	...	
$(vol_b \circ C_{\texttt{int}} \circ \mathit{Rel}_{\texttt{gain}})(n_{\texttt{gain}})$	0	0	1	3	...	

Figure 3.14: Example sequence for `oil-pump` based on composition with clock functions and local indices

Time functions and clock functions for tasks. In addition to the release time function $\mathit{Rel} : \mathbb{N} \to \mathcal{T}$ defined in Section 3.2.2, which denotes the release time of a task for the given activation index $n \in \mathbb{N}$, we define the following additional time functions for tasks:

- $R : \mathbb{N} \to \mathcal{T}$ is the *read time function*, yielding the instant the task reads its values during the current interval. Note that the possibility of R to range over a given time interval incorporates the possibility of multiple reads per activation.
- $W : \mathbb{N} \to \mathcal{T}$ is the *write time function*, yielding the instant the task has last written its values, as seen from the current interval.
- $S : \mathbb{N} \to \mathcal{T}$ is the *start time function*, yielding the instant the task starts running for the current interval.
- $E : \mathbb{N} \to \mathcal{T}$ is the *termination time function*, yielding the instant the task resumes processing and returns control to the scheduler during the current interval.

Clock functions are required to have a discontinuity at each $\mathit{Rel}(n)$: for all $n \in \mathbb{N}$,

$$\begin{aligned} C(\ \mathit{Rel}(n)\ ) &= n \\ \lim_{t \to \mathit{Rel}(n)^-} C(\quad t \quad) &= n-1 \text{ for } t \in \mathcal{T}. \end{aligned}$$

This property will be equivalently stated for functions in Prop. 3.1. The property will make use of a limit operator extended to functions: for a function $F : \mathbb{N} \to \mathcal{T}$, the limit operator $\lim_{F' \to F}$ applied to some expression $G(F)$ is defined as as $\lim\limits_{F' \to F} G(F') = \lambda d\,.\, \lim\limits_{d' \to F(d)} G(d')$. We are then ready to define necessary properties for time and clock functions.

Property 3.1 (Time and clock functions, properties). *We write $Rel^{\ominus}$ for the left limit of Rel, $\lim_{F \to Rel^-} F$. For each individual task, we restrict clock and time functions such that the following properties hold:*

$$
\begin{array}{lrcl}
1. & C \circ X & = & Id \text{ for } X \in \{Rel, R, S, E\} \\
2. & C \circ Rel^{\ominus} & = & Id - 1 \\
3. & Rel \circ C & \leq & Id \\
4. & Rel^{\ominus} \circ C & \geq & Id - 1 \\
5. & Rel & \leq X \leq & Rel^{\ominus} \circ (Id + 1) \text{ for } X \in \{Rel, R, S, E\} \\
6. & W \circ C \circ W & = & W \\
7. & Rel & \leq & W
\end{array}
$$

With the formal framework in place, it can now be demonstrated why preservation alone is not a sufficient criterion for correctness of an implementation in our framework. For instance, a preservation proof alone may assert that for two tasks T_i (writer), T_j (reader), with the same period and offset ($C_i = C_j$), a message read by T_j at index $n_j \in \mathbb{N}$ always corresponds to the message written at T_i at index $C_i \circ W_i \circ C_i \circ R_j(n_j)$. The underlying assumption would be that $W_i \circ C_i \circ R_j(n_j)$ indeed refers to "the last T_i-write instant before the last T_j-read instant, as seen from index n_j". However, this claim rests on the assumption that the "before" assertion indeed holds, and thus $W_i \circ C_i \circ R_j(n_j) < R_j(n_j)$ for all $n_j \in \mathbb{N}$. So in addition to the preservation proof, a proof is needed which demonstrates that read instants R_j always follow corresponding write instants W_i for all possible $n_j \in \mathbb{N}$. In other words, `reads` and `writes` must be consistent with the timing assumptions inherent in the preservation proof.

Definitions and Properties. We shall establish a number of preliminary definitions and properties which will be needed in turn to establish preservation and timing consistency for the three configurations in Theorems 3.6–3.11.

Definition 3.2 (Fixed-priority scheduling, start/finish times). *Let T_i, T_j be two tasks controlled by a fixed-priority, preemptive scheduler. Furthermore, let*

$Prio_i > Prio_j$. If task T_i is released simulatenously to or after task T_j, then T_i finishes before T_j starts.

$$Rel_i \circ C_i \leq Rel_j \circ C_j \implies E_i \circ C_i \leq S_j \circ C_j$$

Property 3.2 (Harmonic clocks, release times). *Let T_i and T_j be two tasks with harmonic clocks $c_i \preceq c_j$. Then*

$$\begin{array}{lrcl} 1. & Rel_i \circ C_i & \geq & Rel_j \circ C_j \\ 2. & Rel_i \circ (Id{+}1) \circ C_i & \leq & Rel_j \circ (Id{+}1) \circ C_j \end{array}$$

Lemma 3.3 (Harmonic clocks, properties). *Let T_i and T_j be two tasks with harmonic clocks $c_i \preceq c_j$. Then*

$$\begin{array}{lrcl} 1. & C_j \circ Rel_i \circ C_i & = & C_j \\ 2. & C_j \circ Rel_i^{\ominus} \circ (Id{+}1) \circ C_i & = & C_j \\ 3. & Rel_i \circ C_i \circ Rel_j & = & Rel_j \end{array}$$

The proof can be found in the appendix, and makes extensive use of Prop. 3.2.

Definition 3.3 (Write sampling). *Task T_i is write sampling with respect to task T_j if, for a given j-period, the write instant of T_i occurs exclusively in the first i-period within the j-period:*

$$Rel_i \circ C_i \circ Rel_j \circ C_j < W_i \circ C_i < Rel_i \circ (Id{+}1) \circ C_i \circ Rel_j \circ C_j.$$

Lemma 3.4 (Harmonic clocks, write sampling). *Let T_i and T_j be two tasks with harmonic clocks $c_i \preceq c_j$, and let task T_i be write sampling with respect to task T_j. Then the following holds:*

$$\begin{array}{lrcl} 1. & Rel_j \circ C_j & < & W_i \circ C_i \\ 2. & C_i \circ Rel_j \circ C_j & = & C_i \circ W_i \circ C_i. \end{array}$$

And if $c_i \succeq c_j$, the following holds:

$$\begin{array}{lrcl} 3. & C_i \circ W_i & = & Id. \end{array}$$

Again, the proof of Lemma 3.4 is found in the appendix, combining properties from Def. 3.3 and Lemma 3.3.

Double and triple buffers. Special care must be taken so that the behavior of double and triple buffers is correctly mirrored by our framework. We therefore extend our set of time and clock functions by a number of functions relating to the use of double and triple buffers.

- $Buf : \mathbb{N} \to \{-1, +1\}$ is the *buffer index function* yielding the currently readable buffer, and is defined as

$$Buf(n) = (-1)^n$$

- $R'_{i \triangleright j} : \mathbb{N} \to (\mathcal{T} \times \{-1, +1\})$ is the *buffer-extended read time function* for a buffer written by T_i and read by T_j. $R'_{i \triangleright j}$ together with C' formalizes a double buffer, where the buffer is switched at T_i's period. $R'_{i \triangleright j}$ maps a j-step index to a pair of j-read time and buffer index, where the index indicates the buffer which was readable when the message was read. $R'_{i \triangleright j}$ is defined as

$$R'_{i \triangleright j} = \begin{bmatrix} R_j \\ Buf_i \circ C_i \circ R_j \end{bmatrix}.$$

- $Rel'_{i \triangleright j} : \mathbb{N} \to (\mathcal{T} \times \{-1, +1\})$ is the *buffer-extended release time function* for a buffer written by T_i and read by T_j. $Rel'_{i \triangleright j}$ together with C' formally captures a triple buffer, where the first stage of the buffer is switched at T_i's period, and the second stage of the buffer is switched at T_j's period. $Rel'_{i \triangleright j}$ maps a j-step index to a pair of j-read time and buffer index, where the index indicates the buffer which was readable when T_j was last released. $Rel'_{i \triangleright j}$ is defined as

$$Rel'_{i \triangleright j} = \begin{bmatrix} Rel_j \\ Buf_i \circ C_i \circ Rel_j \end{bmatrix}.$$

- $C' : (\mathcal{T} \times \{-1, +1\}) \to \mathbb{N}$ is the *buffer-extended clock function*, yielding the step index when the respective buffer was last writable, and defined as

$$C'(t, k) = \begin{cases} C(t) & \text{if } k = -Buf \circ C(t) \\ (Id - 1) \circ C(t) & \text{if } k = Buf \circ C(t) \end{cases}.$$

Fortunately, Lemma 3.5 will demonstrate that the combination of buffer-extended read and release functions with buffer-extended clock functions is easily simulated by the (simpler) time and clock functions originally defined.

Lemma 3.5 (Double and triple buffers, properties). *Let T_i be a writer task and T_j be a reader task such that $c_i \succeq c_j$. Then the following applies:*

$$\begin{array}{lrcl} 1. \quad c_i \succeq c_j \implies & C'_i \circ R'_{i\triangleright j} & = & (Id-1) \circ C_i \circ Rel_j \\ 2. & C'_i \circ Rel'_{i\triangleright j} & = & (Id-1) \circ C_i \circ Rel_j \end{array}$$

Proof. For 1., we note that $R'_{i\triangleright j}$ is written out as $\left[\begin{smallmatrix} R_j \\ Buf_i \circ C_i \circ R_j \end{smallmatrix}\right]$. We argue that the condition $k = Buf_i(t)$ for C'_i in the expression $C'_i \circ R'_{i\triangleright j}$ always holds: both k and $Buf_i(t)$ evaluate to $Buf_i \circ C_i \circ R_j(n)$, where n is the current j-index. Consequently, $C'_i(t,k)$ is always equal to $(Id-1) \circ C_i(t)$, and

$$C'_i \circ R'_{i\triangleright j} = (Id-1) \circ C_i \circ R_j.$$

We now need to show that the $C_i \circ R_j$ is indeed equal to $C_i \circ Rel_j$. We note that, due to $Rel_j < R_j < Rel_j^{\ominus} \circ (Id+1)$,

$$C_i \circ Rel_j \circ C_j \leq C_i \circ R_j \circ C_j \leq C_i \circ Rel_j^{\ominus} \circ (Id+1) \circ C_j,$$

But both $C_i \circ Rel_j \circ C_j$ and $C_i \circ Rel_j^{\ominus} \circ (Id+1) \circ C_j$ are equal for $c_i \succeq c_j$ according to Lemma 3.3, so $C_i \circ Rel_j \circ C_j = C_i \circ R_j \circ C_j$, and hence $C_i \circ Rel_j = C_i \circ R_j$.

For 2., again, the condition $k = Buf_i(t)$ for C'_i holds invariantly, so $C'_i(t,k) = (Id-1) \circ C_i(t)$, and thus

$$C'_i \circ Rel'_{i\triangleright j} = (Id-1) \circ C_i \circ Rel_j$$

□

Preservation and timing consistency. Based on the above lemmas and definitions, the main results of this section are two theorems for each of the three cases outlined in Section 3.2.4, where both the equivalence of ideal and implementation semantics, and timing consistency is shown for each case. We start out with the $c_i \prec c_j$ case formalized by Theorems 3.6 and 3.7, followed by Theorems 3.8–3.9 and 3.10–3.11 for the other two cases, respectively.

Theorem 3.6 (Preservation for $c_i \prec c_j$ and single shared variable communication). *Let T_i and T_j be two tasks such that $c_i \prec c_j$. Then communication through a single shared variable realizes undelayed communication from T_i to T_j, corresponding to the undelayed "A" case.*

Proof. Undelayed "A" case yields the following proof obligation for the preservation proof:

$$C_i \circ Rel_j = C_i \circ W_i \circ C_i \circ R_j.$$

Chaining with C_j, we can rewrite the proof obligation to

$$C_i \circ Rel_j \circ C_j = C_i \circ W_i \circ C_i \circ R_j \circ C_j.$$

Using $C_i \circ W_i \circ C_i = C_i \circ Rel_j \circ C_j$ from Lemma 3.4, we obtain

$$C_i \circ Rel_j \circ C_j = C_i \circ Rel_j \circ C_j \circ R_j \circ C_j.$$

But this is clear since $C_j \circ R_j = Id$. □

Theorem 3.7 (Timing consistency for $c_i \prec c_j$ and single shared variable communication). *Let T_i and T_j be two tasks such that $c_i \prec c_j$. Then communication from T_i to T_j through a single shared variable is timing consistent, corresponding to the properties*

1. $W_i \circ C_i \quad < \quad R_j \circ C_j$ *(read follows write)*
2. $R_j \circ (Id - 1) \circ C_j \quad < \quad W_i \circ C_i$ *(read precedes next write).*

Proof. For showing 1., by assumption, priority assignment is rate monotonic, so $Prio_i > Prio_j$. Chaining Def. 3.2 with $Rel_j \circ C_j$ on both sides and simplifying terms for T_j, we obtain

$$Rel_i \circ C_i \circ Rel_j \circ C_j \leq Rel_j \circ C_j \implies E_i \circ C_i \circ Rel_j \circ C_j \leq S_j \circ C_j$$

It is clear from $Rel_i \circ C_i \leq Id$ that the implication assumption holds. Combining the conclusion with $S_j \circ C_j < R_j \circ C_j$,

$$E_i \circ C_i \circ Rel_j \circ C_j \leq S_j \circ C_j < R_j \circ C_j.$$

From Lemma 3.4, it follows that $W_i \circ C_i \circ Rel_j \circ C_j = W_i \circ C_i \circ W_i \circ C_i$, and thus, using $W_i \circ C_i \circ W_i = W_i$, $W_i \circ C_i \circ Rel_j \circ C_j = W_i \circ C_i$. Combining with $W_i < E_i$, we obtain

$$W_i \circ C_i = W_i \circ C_i \circ Rel_j \circ C_j < E_i \circ C_i \circ Rel_j \circ C_j.$$

Combining the latter two inequations yields

$$W_i \circ C_i < R_j \circ C_j$$

which proves case 1.

For case 2., Lemma 3.4 yields

$$Rel_j \circ C_j < W_i \circ C_i.$$

But also, because of $R_j < E_j$ and $E_j < Rel_j \circ (Id + 1)$,

$$R_j \circ (Id - 1) \circ C_j < E_j \circ (Id - 1) \circ C_j < Rel_j \circ C_j.$$

Combining both inequations yields

$$R_j \circ (Id - 1) \circ C_j < W_i \circ C_i.$$

□

Theorem 3.8 (Preservation for $c_i \succeq c_j$ and double buffer communication). *Let T_i and T_j be two tasks such that $c_i \succeq c_j$. Then communication from T_i to T_j through a double buffer, where the copy is performed at T_i's release times, realizes delayed communication from T_i to T_j,*

$$(Id-1) \circ C_i \circ Rel_j = C_i \circ W_i \circ C'_i \circ R'_{i \triangleright j},$$

corresponding to the delayed single "N" case.

Proof. According to Lemma 3.5,

$$C'_i \circ R'_{i \triangleright j} = (Id-1) \circ C_i \circ Rel_j,$$

so the obligation becomes

$$(Id-1) \circ C_i \circ Rel_j = C_i \circ W_i \circ (Id-1) \circ C_i \circ Rel_j.$$

But $C_i \circ W_i = Id$ according to Lemma 3.4, so the equality follows. □

Theorem 3.9 (Timing consistency for $c_i \succeq c_j$ and double buffer communication). *Let T_i and T_j be two tasks such that $c_i \succeq c_j$. Then communication from T_i to T_j through a double buffer, where the copy is performed at T_i's release times, is timing consistent, corresponding to the properties*

1.	$W_i \circ C'_i \circ R'_{i \triangleright j} < R_j$	*(read follows write)*
2.	$R_j < W_i \circ (Id+2) \circ C'_i \circ R'_{i \triangleright j}$	*(read precedes write after next)*

Proof. For showing 1., we expand $C_i' \circ R'_{i \triangleright j}$ according to Lemma 3.5, yielding

$$W_i \circ (Id-1) \circ C_i \circ Rel_j < R_j.$$

Clearly $W_i \circ (Id-1) \circ C_i < Rel_i \circ C_i$, so it suffices to show

$$Rel_i \circ C_i \circ Rel_j < R_j,$$

which is clear because $Rel_i \circ C_i \leq Id$, and $Rel_j < R_j$.

For showing 2., again using Lemma 3.5,

$$\begin{aligned} R_j &< W_i \circ (Id+2) \circ (Id-1) \circ C_i \circ Rel_j \\ R_j &< W_i \circ (Id+1) \circ C_i \circ Rel_j \end{aligned}$$

We note that $Rel_i \circ (Id+1) \circ C_i < W_i \circ (Id+1) \circ C_i$, so we can show 2. by demonstrating

$$R_j < Rel_i \circ (Id+1) \circ C_i \circ Rel_j.$$

According to Prop. 3.2, $Rel_j \circ (Id+1) \circ C_j \leq Rel_i \circ (Id+1) \circ C_i$, so again it suffices to show

$$R_j < Rel_j \circ (Id+1) \circ C_j \circ Rel_j,$$

which is clear from $C_j \circ Rel_j = Id$ and $R_j \leq Rel_j \circ (Id+1)$. □

Theorem 3.10 (Preservation for possibly unrelated c_i, c_j and triple buffer communication). *Let T_i and T_j be two tasks such that c_i and c_j may be unrelated according to $\preceq$. Then communication from T_i to T_j through a triple buffer, where the first copy is performed at T_i's release times, and the second copy is performed at T_j's release times, realizes delayed communication from T_i to T_j,*

$$C_i \circ W_i' \circ C_i' \circ Rel'_{i \triangleright j} \circ C_j \circ R_j = (Id-1) \circ C_i \circ Rel_j,$$

corresponding to the delayed double "N" case.

Proof. Lemma 3.5 yields

$$C_i' \circ Rel'_{i \triangleright j} = (Id-1) \circ C_i \circ Rel_j.$$

Our obligation can be rewritten to

$$(Id-1) \circ C_i \circ Rel_j = (Id-1) \circ C_i \circ Rel_j \circ C_j \circ R_j,$$

which is clear because of $C_j \circ R_j = Id$. □

Theorem 3.11 (Timing consistency for possibly unrelated c_i, c_j and triple buffer communication). *Let T_i and T_j be two tasks such that c_i and c_j may be unrelated according to $\preceq$. Then communication from T_i to T_j through a triple buffer, where the first copy is performed at T_i's release times, and the second copy is performed at T_j's release times, is timing consistent, corresponding to the properties*

1. $W_i \circ C_i' \circ Rel'_{i \triangleright j} \circ C_j \circ R_j < R_j$ *(read follows write)*
2. $R_j < W_i \circ (Id+2) \circ C_i' \circ Rel'_{i \triangleright j} \circ C_j \circ R_j$ *(read precedes write after next)*

The proof is analogous to the one for Theorem 3.9.

We have thus demonstrated that each of the three configuration rules $c_i \prec c_j$, $c_i \succeq c_j$, $c_i \not\preceq\not\succeq c_j$ from Section 3.2.4 is semantics-preserving, so that the message indices as seen by the reader coincide in the ideal and the implementation semantics, and timing-consistent, meaning that the timing assumptions for the preservation proof are justified. This concludes our implementation scheme for the singleprocessor case, where communication is between tasks of potentially different rates hosted on the same processor. The next section will deal with a strategy for distributing a synchronous dataflow program onto a network of processors.

3.3 Multiprocessor implementation

3.3.1 Introduction

Looking at real-time control application in the automotive sector, time-triggered protocols [Kop97], which have been designed for high levels of fault tolerance and determinism, are an attractive target platform for highly critical applications such as X-By-Wire. Because this kind of architecture fits the time-synchronous abstraction well, distribution of synchronous programs is feasible and has been demonstrated for the synchronous programming language LUSTRE in [CCM+03b]. However, for applications where cost concerns and legacy integration issues are more dominant compared to criticality requirements, existing *event-triggered* bus architectures such as Controller Area Network (CAN) may play an important role for some time to come. Applications characterized by this requirement will be called *medium-criticality applications* in the sequel.

Synchronous approach vs. firm real time. In the context of medium-criticality applications, let us discuss the synchronous distribution issue in more depth: Certainly, semantically correct implementation of the distributed program is vital. But when looking at the state of the art in distributed real-time control applications, many of these applications meet their timing and criticality requirements even though they are based on communication media that provide no absolute guarantees about response times. As a possible explanation, some control applications are known to tolerate the loss of a bounded number of messages, e. g. state values. In real-time systems, this corresponds to the notion of *firm real-time*: transactions are discarded when they miss their deadlines, as there is no value to completing them afterwards. In contrast to hard real-time systems, a bounded number of deadline misses is not considered fatal. How can the notion of firm real-time be married with the distribution of synchronous programs?

It can also be questioned whether synchronous implementations require the existence of a precise global timebase. Existing works on asynchronous distribution of synchronous programs [CGP99] have shown that this is not necessarily the case. On the other hand, it is quite clear that asynchronous distribution does not satisfy some requirements of real-time control applications when communication media are involved that allow message losses or unbounded latencies. Can we use a loose timebase, which may be cheaper to implement, and still obtain implementations suited for real-time control applications?

Based on this discussion, our approach to multiprocessor implementation is twofold:

1. Provide a distribution method for synchronous programs based on a synchronization / communication layer with a loose timebase. The method should ensure semantically correct execution of the synchronous program under normal operation conditions.

2. Make sure the synchronization / communication layer provides a reduced service, including synchronization, in case of certain faults. By adjusting some well-defined parameters, it is then up to the developer to ensure that the system remains in normal operation for the most part of its lifecycle, and meets the correctness and timeliness requirements imposed by the application.

In the following, we will describe the procedure to deploy synchronous programs onto event-triggered networks based on loose synchronization, and evaluate our method with respect to the requirements of medium-criticality applications. The following paragraphs will introduce in detail the concept of a *synchronization cascade*, which provides a layer for synchronization and communication for distributed implementation of synchronous programs.

3.3.2 Synchronization cascades

Terminology. A synchronization cascade provides a layer for synchronization and communication, and implements a logical network topology on top of some suitable physical topology where each link in the logical topology can be mapped to a physical counterpart. We call the underlying protocol(s) the *base protocol(s)* of the cascade.

A synchronization cascade is a rooted tree with nodes $\mathcal{N} = \{N_0, N_1, \ldots\}$, and edges $S \subseteq \mathcal{N} \times \mathcal{N}$. Each node corresponds to a processor or control unit in the distributed implementation. Edges $s \in S$ are called (direct) *synchronizing links*: Each such link communicates a periodic message that is used by its child node to synchronize itself with the parent node.

The root of the tree is called *master node* N_0. For a non-master node N, we denote as $Li(N)$ the set of those synchronizing links that form a path from N_0 to node N. If $(N, N_0) \notin S$, the links in $Li(N)$ form an *indirect link* from N to N_0. $Par(N)$ is the set of *parent nodes* along the path such that $N_0 \in Par(N)$ and $N \notin Par(N)$.

The rooted tree is extended to a (directed) multigraph by adding edges $U = \{u_1, u_2, \ldots\}$, depicted as dashed edges, between nodes. The edges

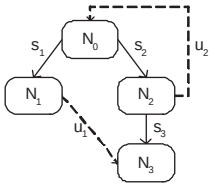


Figure 3.15: Example for a synchronization cascade

$u \in U$ are called *nonsynchronizing links*: while their value is usually important to the receiver, the timing of their reception does not influence the receiver's activation times.

An example for a synchronization cascade is shown in Fig. 3.15: Node N_0 is the master node. Links s_1, s_2, s_3 are synchronizing links, while u_1 and u_2 are nonsynchronizing. The master node emits a periodic synchronizing message with a predefined *base period* T.

Processing phases. Fig. 3.16 shows, schematically, the timing of computations performed by a single node. During each cycle, the node performs two subsequent computations: a *send/receive* phase, and a *computation* phase. For given $N_i \in \mathcal{N}, j \in \mathbb{N}_0$, instant $t_{i,j}$ denotes the activation instant of node N_i at step j.

Send/receive phase. The send/receive phase is triggered by a periodically elapsing timer for the master, and by the respective synchronizing message for a non-master node. During this phase, the nonsynchronizing messages received since the last send/receive phase and the incoming synchronizing message are read, and all outgoing messages computed in the last computation phase are emitted. Because the send/receive phase requires nonzero time for execution, and the receiver node could potentially lose synchronizing messages if their inter-arrival time is too short, we define a *quiet interval* that overlaps the send/receive phase, and during which the node is not required to process incoming synchronizing messages. The remaining part of the cycle is called the *receptive interval*. For nonsynchronizing messages arriving in the quiet interval, the node may either read the message immediately, or leave it in the message buffer so it can be processed by the next send/receive phase. The quiet interval $(0, Q]$ starts at the beginning of each period. The analysis below will ensure that a node does not receive synchronizing messages in $(0, Q]$ under given operating conditions. Q is typically a worst-case estimate of the send/receive period's combined task response and execution times. In the following,

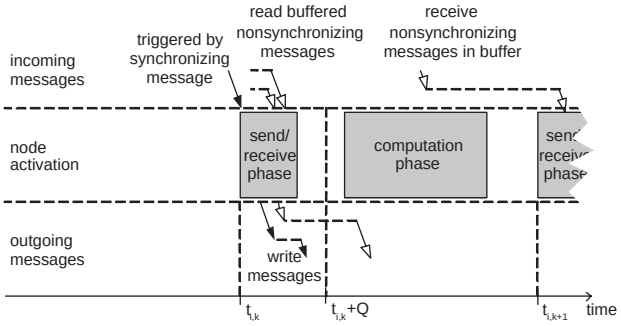


Figure 3.16: Processing phases for step k of a node N_i. Filled arrowheads denote synchronizing messages, empty arrowheads correspond to non-synchronizing messages

we formally require $0 \leq Q < T \cdot (1 - 2\varepsilon)$, where ε is a clock drift constant introduced in Section 3.3.3. Because communication overhead is usually small compared to computation time, we expect typical assignments for Q to be less than $T/4$.

Computation phase. During the computation phase, the local part of the distributed program is executed, the received messages are processed, and the next values of the outgoing messages are computed. Outgoing messages are buffered till the next send/receive phase.

Note that the computation phase may be interrupted by the next send /receive phase under certain circumstances. It is assumed that the send/ receive handler uses default values for all of those outgoing synchronizing messages where no value has been computed in the last step. Consequently, the availability of a synchronization message for the next cycle does not depend on the completion of the computation phase.

Activation of the send/receive phase. Each node N_i defines the following functions and variables:

- getSynchronizingMessage: takes no arguments, and yields the value of the current synchronizing message.
- sendReceive: takes a synchronizing message as a parameter, and executes the send/receive phase based on the value of the synchronizing message.
- getDefaultMessage: takes no arguments, and yields a default message for the synchronizing message, for instance based on the last

available values of the message.

- state $\in$ {EXTERNALLY_TRIGGERED, MESSAGE_ABSENT, SELF_TRIGGERED} is a state variable.

- timer$_i$ $\in \mathbb{R}$ is the physical timer of node N_i.

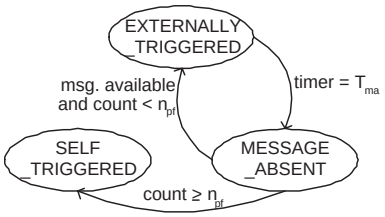
- count $\in \mathbb{N}$ is a counter.

The send/receive phase of each node is initiated by two tasks, message_available_task and timer_task. timer_task is activated T time units and, if necessary, T_{ma} time units after the last activation. timer_task has an idealized release time of zero. The two tasks and the states and transitions of the activation algorithm are shown in Fig. 3.17.

States and transitions. After initialization of the cascade, the master node is in state SELF_TRIGGERED, all other nodes are in state EXTERNALLY_TRIGGERED. In state EXTERNALLY_TRIGGERED, the respective node is synchronized with its parent node, and the send/receive phase is periodically activated by the synchronizing message. In state MESSAGE_ABSENT, the node has detected a (possibly transient) absence of the synchronizing message. The send/receive phase is activated by the node's own periodic timer in this state. We will show in Section 3.3.4 that, while in state MESSAGE_ABSENT, the node is able to re-synchronize itself with its parent node. In state SELF_TRIGGERED, the node is periodically activated by its own timer, and there are no guarantees about the node's ability to re-synchronize itself with its parent node, if existent. The parameter T_{ma} is called the *message absence detection margin*. It denotes the time interval after which, if no synchronizing message has been detected, a node in state EXTERNALLY_TRIGGERED changes to MESSAGE_ABSENT. Parameter n_{pf} is the *parent fault detection count*. It denotes the maximum number of periods the node will remain in state MESSAGE_ABSENT if no synchronizing message is detected. If this number exceeds n_{pf}, the node will change to state SELF_TRIGGERED. Sender fault detection therefore initiates a fallback behavior in case either the parent node or the communication medium fails for a longer period of time. Note that re-synchronization with the parent after the node has entered state SELF_TRIGGERED is not in the scope of this thesis.

```
message_available_task:
  if state ∈ {EXTERNALLY_TRIGGERED,
        MESSAGE_ABSENT} then
    timer_i := 0
    state := EXTERNALLY_TRIGGERED
    sendReceive(getSynchronizingMessage()) endif
timer_task:
  if state = SELF_TRIGGERED then
    if timer_i = T then
      timer_i := 0
      sendReceive(getDefaultMessage()) endif
  else if state = MESSAGE_ABSENT then
    if timer_i = T then
      timer_i := 0, count := count + 1
      if count ≥ n_pf then
        state := SELF_TRIGGERED endif
      sendReceive(getDefaultMessage()) endif
  else
    if timer_i = T_ma then
      timer_i := 0, count := 0
      state := MESSAGE_ABSENT endif endif
```

(a) Send/receive phase activation and transitions



(b) States and transitions

Figure 3.17: Activation, states, and transitions of a node N_i

3.3.3 Environment assumptions

We will now state some assumptions about the physical environment of the cascade. The assumptions will be necessary in order to show that the cascade meets its operational requirements. Some of the assumptions will be required independent of the network state, while others are prerequisites for normal operation of the network, and may be violated under fault conditions.

Physical clocks. Each node N_i has its own physical clock timer_i. A physical clock is typically subject to drifts and jitter w.r.t. the ideal physical time t. Operation of a synchronization cascade requires that deviations of all the nodes' clocks from ideal time are bounded by a constant.

Definition 3.4 (ε-Bounded Clock Drift). *For a given cascade, let each node N_i be associated with a physical clock* timer_i. *The cascade is said to have an* ε-bounded clock drift *iff, for all intervals where* timer_i *is not reset,*

$$\forall N_i \in \mathcal{N}. \left| \frac{d\mathsf{timer}_i}{dt} - 1 \right| \leq \varepsilon$$

In combination with our definition of timer_task, the bounded clock drift assumption guarantees that the physical base period of each node is bounded by $[T/(1+\varepsilon), T/(1-\varepsilon)]$, and the message absence detection period is bounded by $[T_{ma}/(1+\varepsilon), T_{ma}/(1-\varepsilon)]$.

Message jitter. We define for each link s_j, u_j in the cascade a map Δ_{li} mapping direct links, i. e. links between adjacent nodes, to their corresponding worst case *message jitters*, assuming that some adequate method for analysis is available[3]. The minimum and maximum message latencies for direct or indirect links $(N_i, N_{i'})$ will be denoted as $d_{min}(i,i'), d_{max}(i,i')$, respectively, such that

$$d_{max}(i,i') - d_{min}(i,i') = \textstyle\sum_{s_j \in Li(N)} \Delta_{li}(s_j) \tag{3.1}$$

holds for all (N, N').

The message jitter summarizes the end-to-end jitter from the instant the send/receive phase at the parent is activated until the child node's activation time. The worst-case jitter will typically include

[3]For the CAN protocol, the analysis described in [TB94] yields both bounds for worst-case response times and message jitters on the bus.

1. execution time jitter of the sender's send/receive code,
2. queuing jitter at the sender,
3. communication jitter of the medium, and
4. response time jitter of the receiver's task.

Because the message jitter includes the queuing jitter at the sender, the bound may be invalid if the communication medium is not accepting messages (e. g. due to unforeseen overload conditions or external disturbances). We therefore assume the existence of a simple *communication layer* that enforces the predetermined queuing interval by retracting the message when the precomputed worst-case queuing time is overrun. Note that this typically requires the layer to have some access to lower-layer operations of the controller.[4]

For correct operation of the cascade, the end-to-end jitter from the master to any node must be bounded:

Definition 3.5 (Bounded sync. message jitter). *Let $Li(N)$ denote the set of all synchronizing links that form a path from the master to the node N. The network is said to have a* bounded synchronizing message jitter *iff*

$$\forall N \in \mathcal{N}. \sum_{s_j \in Li(N)} \Delta_{li}(s_j) < \min\left(\frac{T-Q}{2}, \frac{T(1-5\varepsilon)}{2}\right)$$

This bound should be satisfiable for a large number of practical applications. For instance, in an automotive case study described in [TB94], for the case of a 1MBit/s CAN bus, most high-priority messages have a jitter of around $10^{-3}s$, so for $\varepsilon = 10^{-6}$, $T = 10^{-2}s$, and $Q = T/20$, cascades up to depth 4 (four synchronizing links between master and the "farthest" node) are possible.

Logical channels. In order to define properties of the communication medium with respect to message loss in the next paragraphs, and to reason about the correctness of the cascade with respect to communication and synchronization in Section 3.3.5.

A stream processing function f, as introduced in Section 2.2.2, is associated with a set of input variables $X_I = \{x_1, x_2, \ldots, x_l\}$ with symbol

[4] In the case of the CAN protocol, the two most popular controller ICs (Intel 82527 and Philips 82C200) allow to retract messages after they have been put in the send buffer

domain V_I, to a set of output signals $O = \{x_{l+1}, x_{l+2}, \ldots, x_n\}$ with symbol domain V_O. $\mathcal{I} = X_I \to V_I^\omega$ and $\mathcal{O} = X_O \to V_O^\omega$ are the input and output domains of the function, respectively. f is assumed to be a total function $f : \mathcal{I} \to \mathcal{O}$. We also employ a special treatment of the message $\perp$, the absent message defined in Section 2.2.1: for a given symbol domain V, we write $V^\perp = V \cup \{\perp\}$ for the set obtained by adding the absent message.

Stream processing functions may be used for defining different classes of logical *channels*, which may be understood as a discrete-time abstraction of some physical reality, e. g. a communication medium, or a combination of communication medium and synchronization cascade.

Definition 3.6 (Channel). *A function f denotes a* channel *iff $|X_I| = |X_O| = 1$ and $V_I = V_O$.*

Definition 3.7 (m-Length-Preserving Channel). *Given some $m \geq 0$, a channel is an m-*length-preserving channel *iff $\forall \sigma_I \in \mathcal{I}.\#(f(\sigma_I)) = \#(\sigma_I) + m$.*

Definition 3.8 (Unit Delay Channel). *A channel is a* unit delay channel with initial message m *iff $\forall \sigma_I \in \mathcal{I}.f(\sigma_I) = m\&\sigma_I$.*

Note that this definition of unit delay channel coincides with the semantics of the `fby` operator defined in Section 2.2.2. As a corollary, unit delay channels are 1-length-preserving.

Definition 3.9 (n-Bounded Lossy Channel). *Given some $n > 0$ and some m-length-preserving channel ch with symbol domains $V_I = V_O = V^\perp$, $\perp \in V^\perp$, ch is an n-*bounded lossy channel *iff, for all input/output streams $(\sigma_I, \sigma_O) \in f$, for all of σ_I's substreams σ_I^i of length n at position i, for all of σ_O's substreams σ_O^{i+m} of length n at position $i + m$, the following condition holds:*

$$\#(\sigma_I^i|_V) = n \implies \#(\sigma_O^{i+m}|_V) \geq 1$$

Intuitively, if fed with messages from the set $V = V^\perp \backslash \{\perp\}$, an n-bounded lossy channel will lose at most $n - 1$ subsequent messages. As a direct consequence of the above definition, any n-bounded lossy channel is also $n + 1$-bounded lossy.

Message loss. The synchronization mechanism has to meet certain fault tolerance requirements. A typical fault in event-triggered real-time systems is the loss of a message: the loss can be caused by the sender when aborting a send operation (e. g. if the queuing delay is longer than expected, and a newer value is available), or by the communication medium itself. Seen more abstractly, we can associate message loss with the *input/output behavior of a link* in the cascade. The following definition will capture this:

Definition 3.10 (I/O function of a (direct) link). *For a given execution of a cascade, the* I/O function of a link $l \in S \cup U$, *written* f_l, *is defined as the function mapping the sequence of messages written by the sender's program to the sequence of messages arriving at the receiver node, where the special output symbol* $\perp$ *indicates a lost message.*

Definition 3.11 (I/O function of an indirect link). *For a given execution of the cascade, the* I/O function of an indirect link $l = l_1 \rightarrow l_2 \rightarrow \cdots \rightarrow l_m$, *where* $l_i \in S \cup U$ *and* l_1 *and* l_m *are the first and last links in the direction of message flow, respectively, is defined as the composition of the individual links' I/O functions:*

$$f_l = f_{l_m} \circ \cdots \circ f_{l_2} \circ f_{l_1}$$

Using these definition, a *lossy* link models both message loss due to the sender's communication layer aborting the send, and due to the medium losing messages. In order to define *bounded* message losses, we will use Def. 3.9 of an n-bounded lossy channel to capture the condition that a cascade may suffer from a bounded number of message losses on each of its direct and indirect synchronizing links.

Definition 3.12 (n-Bounded Lossy Cascade). *A cascade with master* N_0 *is an* n-bounded lossy cascade *iff, for all executions of the cascade, for all direct and indirect synchronizing links from* N_0 *to nodes* $N \in \mathcal{N}$, *the link's input/output function is an* n*-bounded lossy channel.*

Choice of Parameters. We now give some predefined values for the parameters T_{ma} and n_{pf} that can be used by the send/receive phase activation algorithm, and will be incorporated in the analysis of the next section. T_{ma} is chosen such that loss of the synchronizing messages cause the receiver's activations to be delayed by $T/2$ w.r.t. the master's activation instants, while n_{pf} results from an analysis of the maximum number of lost messages that can be tolerated by the synchronization algorithm:

$$T_{ma} = \frac{3}{2}T, \tag{3.2}$$

$$n_{pf} = \max_{N \in \mathcal{N}} \left(\left\lfloor \frac{1}{4\,T\varepsilon} \left(T - \left(2 \sum_{s_j \in Li(N)} \Delta_{li}(s_j) + \max(2Q, T\varepsilon) \right) \right) \right\rfloor \right), \tag{3.3}$$

where the computed value for n_{pf} is required to be positive. For $\varepsilon \ll 1$, $Q \ll T$, the choice for T_{ma} can be shown to be very close to an optimally robust assignment, so that a maximal number of lost synchronizing messages can be tolerated in the presence of clock drifts. We will demonstrate

in the next section that, for our assignment for T_{ma}, the cascade indeed satisfies this robustness requirement for an n_{pf}-bounded number of lost messages.

3.3.4 Analysis of operational modes

This section will provide an analysis of the different operational modes of the cascade: operation under *normal conditions*, operation under *transient fault conditions*, and operation under *permanent fault conditions*. For normal operation, we will show that all non-masters remain in state EXTERNALLY_TRIGGERED, while under transient fault conditions, it will be shown that non-masters never enter state SELF_TRIGGERED.

The following two definitions deal with the property of *synchronization*, which asserts that the offset of the node's activation instant with respect to the master's activation instant is bounded, and *receptiveness*, which guarantees that a node is able to receive the synchronizing message for the current step during its receptive interval. Note that for forwarding a synchronizing message over the entire length of an indirect link, *all* nodes along the link have to be receptive.

Definition 3.13 (j-synchronization). *For a node N_i and a step $j \in \mathbb{N}_0$, the statement "N_i is j-synchronized" corresponds to the property*

$$d_{\min}(0,i) \leq t_{i,j} - t_{0,j} \leq d_{\max}(0,i) \tag{3.4}$$

Nodes in $\mathcal{N}$ are assumed to be 0-synchronized (proper initialization of the cascade). Furthermore, we define that the master node N_0 is j-synchronized for all $j \in \mathbb{N}_0$. Again, j-synchronization is extended to sets of nodes and indices.

Definition 3.14 (j-receptiveness). *For a node N_i and a step $j \in \mathbb{N}$, the statement "N_i is j-receptive" corresponds to the three properties*

$$t_{0,j} + d_{min}(0,i) \;>\; t_{i,j-1} + Q, \tag{3.5}$$

$$\begin{aligned}(j-1)\text{−synchronized}(N_i) \implies \\ t_{0,j} + d_{max}(0,i) \;<\; t_{i,j-1} + \frac{T_{ma}}{1+\varepsilon},\end{aligned} \tag{3.6}$$

$$\begin{aligned}\neg(j-1)\text{−synchronized}(N_i) \implies \\ t_{0,j} + d_{max}(0,i) \;<\; t_{i,j-1} + \frac{T}{1+\varepsilon},\end{aligned} \tag{3.7}$$

where $T_{ma}/(1+\varepsilon)$ is the minimum message absence detection period defined in Section 3.3.3. The master node N_0 is defined to be j-receptive for all $j \in \mathbb{N}$. j-receptiveness is extended to sets of nodes $\mathcal{N}' \subseteq \mathcal{N}$ and to sets of indices $J \subseteq \mathbb{N}$

such that $\mathcal{N}'$ is j-receptive iff all of its members are, and N_i is J-receptive iff N_i is j'-receptive for all members $j'' \in J$.

Definition 3.15 (Normal operating conditions). *A cascade is said to be under* normal operating conditions *iff it is 1-bounded lossy, and the ε-bounded clock drift assumption holds.*

Lemma 3.12. *Let N_i be a non-master node in a cascade. Then j-receptiveness of N_i and arrival of a synchronizing message in step j at N_i imply j-synchronization of N_i.*

Proof. By Definition 3.14, it follows from j-receptiveness of N_i that, if a synchronization message originating from N_0 arrives at N_i for step j, it is received during the receptive interval of N_i, leading to an activation of N_i at $t_{i,j}$. According to our assumption about the communication medium, the difference $t_{i,j} - t_{0,j}$ is then bounded by $[d_{\min}(0,i), d_{\max}(0,i)]$, so j-synchronization holds for N_i. □

Lemma 3.13. *Let N be a non-master node in a cascade under normal operating conditions. Then j-receptiveness of N and $Par(N)$ imply j-synchronization of N.*

Proof. Straightforward adaptation of Lemma 3.12: Normal operating conditions ensure that the direct or indirect synchronizing link to N is 1-bounded lossy, so synchronizing messages from N_0 are never lost. Because N_0 will send a message in every step j, and both N and N's parent nodes are j-receptive, a synchronizing message will be received by N in its receptive interval for all j. □

Lemma 3.14. *Let N be some non-master node in a cascade under normal operating conditions. Then $(j-1)$-synchronization of N implies j-receptiveness of N.*

A detailed proof is given in the appendix. Intuitively, it suffices to show that, given j-receptiveness of $Par(N)$ and $j-1$-synchronization of N, messages will always arrive after the quiet interval has elapsed at N, and before timer of N reaches T_{ma}. It is sufficient to examine two corner cases, where (1) N_0's clock is "fast", N's clock is "slow", the synchronizing message at $j-1$ has maximum latency, and the synchronizing message at j has minimum latency, and (2) N_0's clock is "slow", N's clock is "fast", the synchronizing message at $j-1$ has minimum latency, and the synchronizing message at j has maximum latency.

Lemma 3.15. *Under normal operating conditions, all nodes in $\mathcal{N}$ are j-receptive and j-synchronized for all j.*

Proof. Double induction over the index set for j, and over the nodes on the path from N_0 to given node $N \in \mathcal{N}$, using Lemmas 3.13 and 3.14. □

Theorem 3.16. *Under normal operating conditions, a non-master node N will always remain in state* EXTERNALLY_TRIGGERED.

Proof. We observe that N is initialized to state EXTERNALLY_TRIGGERED. Because of Lemma 3.15, N is j-receptive for all j, we conclude from Definition 3.14 that the precondition for leaving state EXTERNALLY_TRIGGERED, $\mathsf{timer}_i = T_{ma}$, will never hold. Therefore, the only reachable state for N is EXTERNALLY_TRIGGERED. □

In case of temporary message losses, the network is operating under *transient fault conditions*: nodes affected by loss of their synchronizing message may transition temporarily to state MESSAGE_ABSENT. We can show, however, that a given node will always re-synchronize itself with the master, and count never reaches n_{pf}. As a consequence, the node will never enter state SELF_TRIGGERED.

Definition 3.16 (Transient fault conditions). *A cascade is said to operate under* transient fault conditions *iff it is n_{pf}-bounded lossy, and the ε-bounded clock drift assumption holds.*

Lemma 3.17. *Let N be some non-master node in a cascade under transient fault conditions. Then if there exists an n, $1 \leq n \leq n_{pf}$, such that N is $(j-n)$-synchronized, then N is j-receptive.*

Again, the details of the proof can be found in the appendix. It demonstrates that, for any n' such that $1 \leq n' \leq n_{pf}$, if a node N has performed $n'-1$ unsynchronized cycles, synchronizing messages will arrive after the quiet interval has elapsed, and before timer of N reaches T. Similarly to Lemma 3.14, the two corner cases are: (1) N_0's clock "fast", N's clock "slow", synchronizing message at $j-n$ with maximum latency, synchronizing message at j with minimum latency, and (2) N_0's clock "slow", N's clock "fast", synchronizing message at $j-1$ with minimum latency, synchronizing message at j with maximum latency.

Lemma 3.18. *Let N be some non-master node in a cascade under transient fault conditions. Then for a given step $j > n_{pf}$, let $J = \{j - n_{pf}, \ldots, j-1\}$ be a set of successive step indices. If N and Par(N) are J-receptive, then there is at least one $j'' \in J$ such that N is j''-synchronized.*

Proof. Transient fault conditions imply that the synchronizing link from N_0 parent to N is n_{pf}-bounded lossy. According to our definition for the node's behavior, N_0 will send a synchronization message in every step. Then from Definition 3.12, it is clear that N receives a synchronizing message for at least one $j'' \in J$, and so Lemma 3.18 is a direct adaptation of Lemma 3.12. □

Lemma 3.19. *Under transient fault conditions, all nodes in $\mathcal{N}$ are j-receptive for all j.*

Proof. The proof is again by double induction over js in the index set, and over the nodes on the paths from N_0 to nodes $N \in \mathcal{N}$.

(1 – base case) $\forall j.j$-receptive(N_0):
By Definition 3.14.

(2 – induction step) $(\forall j.j$-receptive$(\text{Par}(N))) \implies (\forall j.j$-receptive$(N)))$:
Split into cases (2a) and (2b) for the inner induction.

(2a – base case)
$\forall j \leq n_{pf}.j$-receptive(N):
By Definition 3.13, N is 0-synchronized. So there exists an $n \leq n_{pf}$ such that N is $(j-n)$-synchronized and, By Lemma 3.17, N is j-receptive.

(2b – induction step)

$$\begin{aligned} \forall j > n_{pf}.\ & (\{j-n_{pf},\ldots,j-1\}\text{-receptive}(\text{Par}(N)) \\ & \wedge\ \{j-n_{pf},\ldots,j-1\}\text{-receptive}(N)) \\ \implies\ & j\text{-receptive}(N) : \end{aligned}$$

Because N and Par(N) are j'-receptive for all $j' \in \{j-n_{pf},\ldots,j-1\}$, we know by Lemma 3.18 that there exists an $n \leq n_{pf}$ such that N is $(j-n)$-synchronized. But this is just the precondition for Lemma 3.17, so N is j-receptive. □

Theorem 3.20. *Under transient fault conditions, a non-master node will never enter state* SELF_TRIGGERED.

Proof. By Lemma 3.19, the non-master node will be always receptive when receiving a synchronizing message. Therefore, each arriving synchronizing message is received. Transient fault conditions guarantee that the medium loses at most $n_{pf}-1$ subsequent messages. Consequently, count never reaches n_{pf}, so the precondition count $\geq n_{pf}$ for transitioning to state SELF_TRIGGERED will never hold. □

We denote as *permanent fault conditions* all other operating conditions, such as non-n_{pf}-bounded lossy cascades, violation of the ε-bounded clock

drift assumption, or complete failure of nodes. Behavior of the cascade under such conditions will not be discussed in the scope of this thesis.

3.3.5 Properties of synchronization cascades

This section defines some essential requirements that a synchronization cascade has to satisfy for distribution of synchronous programs in medium-criticality applications, and demonstrates the corresponding formal properties.

Requirements. Intuitively, the cascade should fulfill two classes of requirements: *reactivity*, that is, there should be a least frequency at which local processing is performed independent of communication failures, and *channel preservation*, which is related to preservation of communication and synchronization inherent in the original synchronous dataflow program.

P-reactivity Distributed real-time control applications typically contain *periodic, reactive* parts which continually compute output values from a given input. Because inputs may originate from other nodes, it is highly desirable to provide an architecture that allows reactive programs to safely synchronize their local processing with communication on the medium, eliminating the needs for special "watchdogs" or similar mechanisms. We will define a property called *P-reactivity* which captures the fact that a node performs communication actions with a certain minimal frequency. Local processing can therefore be triggered by the communication handler

Definition 3.17 (P-Reactivity). *A node N_i is called P-reactive for some $P \in \mathbb{R}_+$ iff, for all possible executions of N_i and for all instants t, there is at least one activation instant for a send/receive phase in the time interval $[t, t + P)$.*

Unit delay and length preserving channels Semantically correct deployment of a synchronous specification warrants that the communication channels provided by the communication layer are valid implementations of the corresponding abstract channels in the specification. As will be justified by Property 3.22, we will use unit delay channels as our model for an abstract channel. Breaking down the original requirements for the cascade to the implemenation of channels, the cascade should (1) provide a valid implementation of unit delay channels under normal operating conditions, and (2) provide some limited service,

including synchronization, under transient fault conditions. The synchronization service can be abstracted as a lossy channel with the 1-length-preserving property. Length-preservation then captures the fact that sender and receiver never get "out of sync".

Properties. In the following, we will show that the cascade indeed satisfies the stated requirements.

Property 3.21. *Under normal operating conditions or transient fault conditions, all nodes are* $(T_{ma}/(1-\varepsilon))$*-reactive.*

By definition of timer_task in Fig. 3.17, N_i is activated at least every T_{ma} time units (measured by its physical clock) in all of the three possible states EXTERNALLY_TRIGGERED, MESSAGE_ABSENT, SELF_TRIGGERED. Because the bounded clock drift assumption holds under normal and transient fault conditions, the worst case of a "slow" physical clock is $d\mathsf{timer}_i/dt = 1-\varepsilon$. In physical time, the greatest interval in between activations is therefore $T_{ma}/(1-\varepsilon)$. □

Property 3.22. *Under normal operating conditions, the input/output behavior of a synchronizing link is a unit delay channel.*

For an intuitive treatment, there are four parts constituting the unit delay channel property: (1) every message sent by the sender must be accepted by the communication medium, (2) once accepted, the message must reach the receiver, (3) the cascade receiver must be receptive, (4) a message computed at step j at the sender is processed at step $j+1$ at the receiver. (1) and (2) are guaranteed by the 1-bounded message loss assumption. (3) and (4) are direct consequences of Lemma 3.15.

Property 3.23. *Under normal operating conditions or transient fault conditions, the input/output behavior of a synchronizing link is a* 1*-length-preserving channel.*

In the case of normal operating conditions, the 1-length-preserving property results from Property 3.22. For transient fault conditions, the fact that the synchronizing link itself is 1-length-preserving follows from Lemma 3.18: let N, N' be the sender and receiver of the synchronizing message, respectively. For a given step j, there are two possibilities: (1) if the synchronizing message is not lost in step j, then N and N' will both be j-synchronized. They will therefore agree on the step number, and N' will process in step j the result of N's computation at step $j-1$, so the channel's

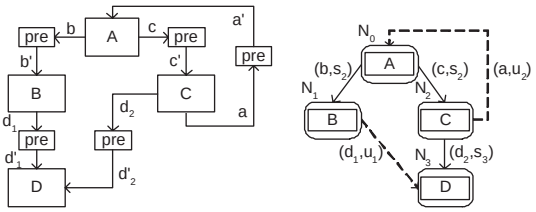
behavior may be characterized as a unit delay for step j. (2) if the synchronizing message is lost in step j, then N' will detect a $\perp$ symbol each time N emits a synchronizing message. In both cases, the input/output behavior of the link constitutes a 1-length-preserving channel.

Nonsynchronizing messages In the discrete-time abstraction of the synchronous programs, synchronizing messages correspond to messages with *deterministic* timing: if the sender component has computed the synchronizing message at step j, the synchronizing message will always be processed by the receiver component at step $j + 1$ in normal operation. This is why abstracting the link as a (deterministic) unit delay channel in the synchronous program, as shown in the Section 3.3.5, is justified for synchronizing links.

For nonsynchronizing links, the deterministic delay channel abstraction may not always be valid. A nonsynchronizing message computed in step j by a sender node may reach a receiver node at steps $j, j+1, \ldots$, depending on the timing of activations of the two nodes, and the timing of messages on the synchronizing link. Fortunately, best/worst-case analysis, such as in [TB94], can be used in theory to ensure that *some* nonsynchronizing messages have deterministic timing. For other messages, it may be necessary to either add some flow control mechanism to the communication layer, or to account for the nondeterminism in the discrete-time abstraction.

Mapping Synchronous Programs to Cascades Consider the simple network in Fig. 3.18(a): The network includes components $\{A, B, C, D\}$, and unit delay channels *pre* in between components. A sends messages through signal b to component B, and through signal c to component C. B sends messages to D (signal d_1), C sends messages through signal d_1 to component D, and to A through signal a. The dataflow network is mapped to the cascade of Fig. 3.15 with the mapping $\{(A, N_0)$, (B, N_1), (C, N_2), $(D, N_3)\}$ Fig. 3.18(b) shows a depiction of the resulting cascade. The resulting distribution is correct for normal operation if the two nonsynchronizing channels u_1, u_2 have deterministic unit-delay behavior. Note that the synchronization messages carry values that the distributed program needs to communicate. If, for a given system step, no such value needs to be communicated, an empty message with no relevant data must be used as a synchronizing message.

Methodical handling of delays. Our method of distribution relies on the existence of delays at the partitioning boundaries in the specification, sim-



(a) Dataflow network

(b) Deployed dataflow network

Figure 3.18: Mapping a dataflow network to a cascade

ilar to the Giotto language [HHK01]. Clearly, introduction of such delays is somewhat implementation-driven: an "ideal" platform with infinite resources would not require delays in the model, except for breaking causal loops. Top-down, implementation-independent development will typically not result in an appropriate placement of delays at partition boundaries. On the other hand, bottom-up introduction of delays will in most cases necessitate a re-validation of the entire design, and is thus in conflict with the idea of having implementation-independent synchronous specifications. We have, however, proposed in [BBR+05] a methodology which enforces introduction of delays at the boundaries of (abstract) software components at early design stages, thus ensuring both implementation-independence and partitionability of the specification.

3.4 Related work

Singleprocessor implementation The work on singleprocessor implementation has originally been done for an internal project report sponsored by ETAS. The results further elaborated in Section 3.2 have originally been published in [BR04]. The implementation scheme has been implemented as part of a translator from AutoFOCUS [HSE97] to ASCET [ETA] as a result of the AutoMoDe project [BBR+05]. Similar work has been published in [HHK01], [SC04], and [TSSC05].

- [HHK01] uses "logical execution times" universally on the level of language, which can be regarded as isomorphic to delayed-only communication in SSDL. Their solution allows communication between arbitrary (non-harmonic) periodic multirate tasks, and is suited for

different schedulers, such as fixed-priority and EDF. As a drawback of the technique, a triple buffer for each communication link is required, equivalent to the $n_i \not\preceq \not\succeq n_j$ case in our framework. Optimizations based on case distinctions comparable to our technique are not considered. Furthermore, all communication is delayed-only.

- In comparison to [SC04], our framework ties the high-to-low priority case to clocks related by $\preceq$. For this case, our method then allows the optimization of using a single shared variable instead of the double buffer used for generic high-to-low communication as in [SC04]. The remaining high-to-low scenarios would be subsumed in our method in case 3, and would incur a unit delay. In contrast to the additional flag-triggered buffer switch algorithm used in [SC04], our method assumes that the appropriate write sampling primitives, which exist on the SSDL program level in the form of `hold` and `every` primitives, are synthesized into the code.

- For the high-to-low case, our single-buffer optimization has subsequently be described in [TSSC05] as part of a more general algorithm.

Multiprocessor implementation Our work on multiprocessor implementation has been previously published in [RB04]. Related works include the LTTA approach of Benveniste et. al. [BCG+02], Time-Triggered CAN (TTCAN) [FMHH01] of Robert Bosch GmbH, and clock synchronization algorithms, as exemplified by Welch's and Lynch's algorithm [WL88].

- In comparison to LTTA [BCG+02], synchronization cascades are designed for protocols with restricted availability, while in LTTA, the bus is assumed to be ideally available. LTTA links may be abstracted as deterministic channels, while in cascades, even under normal conditions, some unsynchronized links may exhibit nondeterministic behavior. In cascades, the activation timing of non-master nodes always depends on the master node, and is roughly periodic. This is also true if synchronous programs with (delayed) feedback loops are deployed onto a cascade. For LTTA networks, convergence of the activation frequencies for length-preserving programs with feedback is not obvious from [BCG+02] [5].

[5] Consider a synchronous program with delayed feedback loop deployed onto an LTTA with nodes N_1, N_2: If the sequence sent by N_1 has periodic timing, the timing of the received, decoded sequence at N_2 is usually aperiodic because N_2's alternating bit decoder will occasionally drop duplicate messages. For providing a periodic feedback

- For the special case of the CAN protocol, TTCAN [FMHH01] is a CAN-based synchronization layer which, in its Level 1 stage, does not rely on additional hardware for synchronization, similar to cascades. Arbitration in TTCAN is primarily based on static assignment of message slots. An interesting question is whether existing unsynchronized nodes can be integrated with the synchronized network: In TTCAN, unsynchronized nodes may only have read access to the bus, while in a CAN-based cascade, full read/write interoperation is possible if the messages sent by unsynchronized nodes are included in the message jitter analysis in Section 3.3.3.

- High-precision clock synchronization algorithms such as [WL88] are well-studied. This kind of algorithm provides a high-precision synchronization, where clocks are synchronized within an interval in the range of $\Delta(1 - 1/|\mathcal{N}|)$, with $|\mathcal{N}|$ the number of nodes, while cascades are merely synchronized in the range of $T/2$, with T as the base period. However, we claim that our design of a synchronization cascade is more specifically suited to the requirements outlined in Section 3.3.1. For instance, Welch and Lynch's algorithm requires $|\mathcal{N}|^2$ synchronization messages for each round vs. $|\mathcal{N}| - 1$ messages for a cascade. Welch and Lynch's algorithm uses explicit synchronization rounds, where the synchronization messages could potentially block other real-time traffic on the medium. Cascades, on the other hand, provide synchronization using the regular real-time traffic of the distributed program. We also claim that the $T/2$ precision may be sufficient in cases where synchronization is important for correct, timely implementation of the distributed program's semantics, but a precise absolute global time base is not necessary.

sequence to N_1, N_2 must adjust its send rate to the average frequency of the decoded sequence, and vice versa for N_1. It is unclear how the frequencies of N_1 and N_2 converge in such cases.

Chapter 4

Linearizations and Property Preservation

This chapter will frame the concrete, application-oriented implementation strategies of Chapter 3 with a more general theory for implementation refinement and property preservation for the synchronous dataflow programs introduced in Chapter 2. The theory is based on the assumption that a synchronous dataflow program is partitioned into a number of subprogram instances, each subprogram instance corresponding to an independent thread in the implementation, where threads communicate through bounded channels. The theory is intended both for broadening the understanding of correct-by-construction implementation schemes, and for possible use in a bottom-up fashion, such as monitoring, testing, or verification. A possible application of the theory in the context of temporal logics for behavioral specification will be highlighted.

It is clear that a physical implementation of a synchronous dataflow program operates in dense, physical time. Imagine that a trace of all communication events is observed for such a running system, where events are ordered according to their relative timestamps. We assume that the distributed timebase for recording event traces is faithful in the sense that, if two events can only appear in a certain order, then this is always reflected in our observation. We also adopt the (frequent) assumption that in physical time, no two events are exactly simultaneous, so the trace is indeed a totally ordered sequence of atomic events.

Based on our knowledge about the original synchronous program, we may want to evaluate such traces with respect to some aspect of interest, e. g. an invariant property that the system has to satisfy. This chapter deals with such properties, how they can be verified on a synchronous abstraction of the system, and how a verification result on the abstract level can

be used for reasoning about the implementation.

- On the abstract level, our formal vehicle will be SSDL programs, their corresponding *synchronous automata*, and *synchronous words*, wherein the word-based semantics trivially corresponds to the stream-based semantics introduced in Section 2.2.2.

- On the implementation level, we shall adopt the notion of *linearized automata*, and their corresponding runs, *linearized words*. Linearized words are sequences of atomic symbols spread out in time, and therefore reflect the assumption that in a physical implementation, no two events happen exactly at the same time.

The rest of this chapter is structured as follows: The notion of linearization, which is used to formalize real-time implementations of synchronous programs, is introduced in Section 4.1. It is shown how the implementation of a synchronous program can be understood as the composition of linearized automata, and how the language of its implementation can be formalized as linearized words. Both formalizations are shown to be equivalent. Section 4.2 puts the linearization theory to work in the framework of Linear Temporal Logic (LTL) property specification. It is shown how LTL properties may be formulated on the level of synchronous programs, how a property can be re-interpreted on the level of linearized implementations, and how the preservation of an LTL property between a synchronous program and its linearization can be asserted using a simple check. Section 4.3, finally, relates this work to other existing work on property preservation, and linearization of synchronous programs.

4.1 Linearizations

4.1.1 Overview

Our linearization theory describes, in a general way, the principal implementation concepts behind the two concrete implementation schemes described in Chapter 3. We shall see that, for the software-based systems which are the subject of this thesis, an implementation of a synchronous program can be described as a combination of *threads* and bounded *channels*.

Threads and channels. For illustration, let us go back to Section 3.2: Each SSDL subprogram instance is implemented as sequential thread, com-

municating with its surrounding threads through single-variable or double/triple buffers, which can be understood as bounded FIFO channels. Flip to Section 3.3: here, an SSDL program again is split into sequential threads running on individual nodes, where each thread computes its results during the computation phase. Communication in synchronization cascades is through logical FIFO channels, which are in effect realized by the physical base protocol, and whose size is bounded by aligning the sender and receiver thread executions through a synchronization mechanism (the cascade). It shows that threads and bounded FIFO channels are indeed the essential ingredients of our theory:

Threads. During an execution, a thread consumes and produces a totally ordered sequence of symbols consistent with the variable dependency relation $\rightsquigarrow^*$, as outlined in Section 3.2.3. For each variable mapped to the thread, exactly one symbol is consumed or produced. These assumptions correspond to the sequential code synthesis scheme outlined in Section 3.2.3. Furthermore, thread executions are assumed to be strictly consecutive: one execution is never interrupted by or interleaved with another execution of the same thread. It is important to keep in mind that, even though threads are parallel units of computation like dataflow processes, thread operational semantics is not always coincident with pure dataflow operational semantics [Kah74]. For instance, in its dataflow interpretation, the SSDL program

```
y := 0 fby 0 fby x
```

may initially produce two symbols for `y`, $\texttt{0}^{\texttt{y}}\texttt{0}^{\texttt{y}}$, before requiring an `x` symbol to proceed. When mapping the above program to one thread, however, the thread operational semantics warrants that two consecutive emissions of `y` symbols be always interleaved with the consumption of an `x` symbol. Consequently, the word $\texttt{0}^{\texttt{y}}\texttt{0}^{\texttt{y}}$ is not accepted by the thread.

Channels. Channels relate two variables `x`, `y`, and are taken to be strictly FIFO, without loss or manipulation of messages. Channels are also assumed to be bounded with a known bound $k \in \mathbb{N}$, as they would not be implementable otherwise. We shall denote a channel of capacity k as a k-channel. Note that there are various implementation mechanisms to achieve boundedness of a communication channel, such as, for example:

- Channels may *write block* when full, forcing the writer thread to block until the reader thread has consumed at least one symbol.

- For each bounded channel, an *acknowledgment channel in the opposite direction* may be introduced, effectively forcing synchronization of writer and reader in a way equivalent to write blocking [Par95].
- The boundedness assertion may be based on an *overall synchronization mechanism*, such as singleprocessor threads synchronized and controlled by a local scheduler as in Section 3.2, the synchronization cascade from Section 3.3, or time-triggered networks [Kop97].
- Even in the absence of strict synchronization, the boundedness assertion may be based on a *stochastic analysis*, taking into account stochastic patterns of writer activations vs. reader availability. This kind of strategy is sometimes encountered in telecommunications or Internet-based systems [Tan96].

As our theory captures the general idea of boundedness rather than a specific means to achieve it, all of the above are covered by the notion of a bounded channel. For further illustration, we will examine a typical mapping of an SSDL program to threads and bounded channels in a first example.

Example (A really simple program and its linearization). Consider the simple SSDL program `twice-identity` shown in Fig. 4.1. Program `twice-identity` has variables $\mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}$, where `w` and `y` are inputs, and dependencies $\mathtt{w} \rightsquigarrow^* \mathtt{x}$, $\mathtt{y} \rightsquigarrow^* \mathtt{z}$. When run, the program will produce individual streams for each variable: an example synchronous run for `twice-identity` is shown in Fig. 4.3. In the figure, we use a specific notation for denoting variable/value pairs, or *symbols*: for the case of integer values, symbols are written as $1^{\mathtt{x}}, 2^{\mathtt{x}}, \ldots, 1^{\mathtt{y}}, 2^{\mathtt{y}}, \ldots$. A synchronous run of `twice-identity` can be thought of as producing subsequent symbol sets, or *macrosymbols*, such as $\{1^{\mathtt{w}}, 1^{\mathtt{x}}, 1^{\mathtt{y}}, 1^{\mathtt{z}}\}\{2^{\mathtt{w}}, 2^{\mathtt{x}}, 2^{\mathtt{y}}, 2^{\mathtt{z}}\}\ldots$. The grouping to macrosymbols in the synchronous run of Fig. 4.3 expresses the fact that the relative position of a symbol within the macrosymbol does not say anything about causal order, while the relative position of the macrosymbol within the word does.

A distribution scheme will implement individual subprogram instances $\mathtt{id}_{\mathtt{x}}$ for computing `x` and $\mathtt{id}_{\mathtt{z}}$ for computing `z`. For instance, this can be done either by assigning one thread to each identity function, or by using one thread for the entire program `twice-identity`, as shown in Fig. 4.2. In each respective implementation, the threads communicate with their environment through bounded channels. We choose channels of capacity one, such that a symbol has to be read from the channel

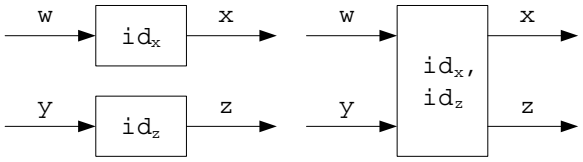
```
program twice-identity;
  input  w,y;
  output x,z;

  program id;
    input in;
    output out;

    out := in;
  endprogram;

  x := id(w);
  z := id(y);
endprogram;
```

Figure 4.1: Program `twice-identity`



(a) Two-threaded implementation (b) One-threaded implementation

Figure 4.2: Two implementations of `twice-identity`

before the next one can be written. For this case, Fig. 4.2(a) illustrates the two-threaded implementation of `twice-identity`, while Fig. 4.2(b) illustrates a one-threaded implementation.

We assume that the implementation is observed in a faithful manner, so the order of observed events coincides with the relative timing of their occurence. Fig. 4.4 shows two example runs for implementations of `twice-identity`, where dashed vertical lines indicate subsequent observations in physical time, and solid lines connect the symbols corresponding to a common index in the synchronous run. The linearized word is shown at the bottom of the respective figure. The one-threaded implementation synchronizes $\mathtt{id}_x$ and $\mathtt{id}_z$, and therefore will not accept the run shown in Fig. 4.4(a). The run in Fig. 4.4(b) reflects a more tightly synchronized program and is accepted by both implementations.

Note that the bounded input channels for `w` and `y` ensure some synchronization between symbols for mutually dependent variables: for instance, there cannot be arbitrarily many `w` symbols before an `x` symbol is produced. However, for mutually independent variables, such as `w` and `y` for the two-threaded implementation, there will not necessarily be any synchronization. Two variables are mutually independent if neither the program itself (through the variable synchronization relation $\xrightarrow{(\cdot)}{}^{*}$, Def. 2.10) nor the chosen implementation structure provide any synchronization between the variable pair.

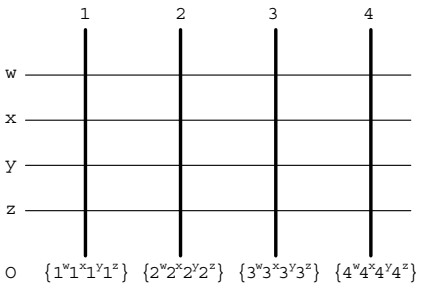
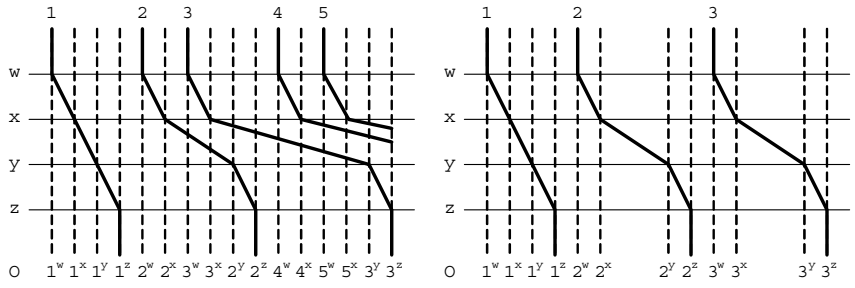


Figure 4.3: Example synchronous run for program `twice-identity`



(a) Two-threaded implementation

(b) One- and two-threaded implementation

Figure 4.4: Example linearized runs for program `twice-identity`

4.1.2 Synchronization and composition

The example of `twice-identity` has shown that the synchronous program's data dependencies, the chosen mapping to threads, and the chosen channel capacity all have an influence on the allowable linearized behaviors of an implementation. We shall capture these different influences on synchronization more formally in this section.

Variable synchronization vs. queuing capacity. It is an important insight that synchronization constraints may stem both from the properties of the *program*, and from the properties of the chosen *implementation*.

Program-induced synchronization. As an example for a program-induced constraint, the immediate dependency between variables `w` and `x` in program `twice-identity` results in the fact that, for any run of `twice-identity`, the number of `x` symbols is always less or equal to the number of `w` symbols, while the number of `w` symbols is not constrained by the program. Such a synchronization property of the program is formally captured in the variable synchronization relation $\xrightarrow{(.)}{}^{*}$, as defined in Def. 2.10. Intuitively, $\xrightarrow{(.)}{}^{*}(\mathtt{x},\mathtt{y}) = n$ incurs that, in a linearized run, `y`'s symbol count may lead `x`'s symbol count by at most n if purely program-induced constraints are considered. For our example of `w` and `x`, $\xrightarrow{(.)}{}^{*}(\mathtt{w},\mathtt{x}) = 0$ and $\xrightarrow{(.)}{}^{*}(\mathtt{x},\mathtt{w}) = \infty$.

Implementation-induced synchronization. As an example for an implementation-induced constraint, we note that for a pair of variables (such as `w` and `x` in program `twice-identity`) that is processed by the same thread, the number of symbols of either variable may lead the symbol count for the other variable by at most one. For instance, by this constraint, $2^{\mathtt{w}}2^{\mathtt{y}}$ is a accepted by the one-threaded implementation of `twice-identity`, but $1^{\mathtt{w}}2^{\mathtt{w}}1^{\mathtt{y}}$ is not. To capture the properties of a given implementation mapping, we define a *queuing capacity relation* $\Delta \subseteq (X \times \mathbb{N}_0 \times X)$ which relates pairs of variables, formally captured in Def. 4.1 below. For the `w` and `x` in `twice-identity`, $(\mathtt{x}, 1, \mathtt{y}) \in \Delta$ and $(\mathtt{y}, 1, \mathtt{x}) \in \Delta$ expresses the desired constraint.

The definition of the queuing capacity relation is based on a partition of an SSDL program to a composition of threads and atomic 1-channels. The nature of this partition will be discussed before giving the actual definition of queuing capacity.

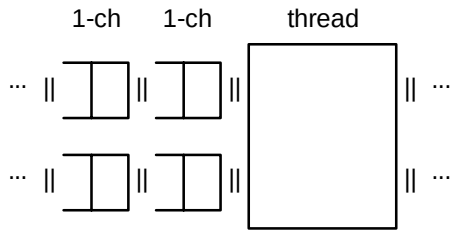


Figure 4.5: Composition of threads and 1-channels (schematic)

Linearizations as compositions of threads and channels. We summarize the mapping to threads and channels for synchronous programs, which will be laid out more completely in Section 4.1.4. Threads and channels can be mutually composed using a composition operator $\|$. A channel with capacity k can be understood as the composition of $k-1$ sequentially combined 1-channels, composed with the receiver thread, which provides one buffering location by default. For instance, a one-threaded implementation of `twice-identity` with 3-channels for `w` and `y`, respectively, is schematically illustrated in Fig. 4.5. Communication between threads and channels is achieved by mutually shared input/output variables. We note that some of the variables used for inter-channel communication may not exist in the original program, so some additional variables need to be introduced.

We assume that an SSDL program is in partitioning normal form (PNF), as described in Section 2.2.3. The PNF structure of an SSDL program does not always match the implementation structure with threads and channels: for a variable $\mathtt{x} \in X$, 1-channels may require additional variables $\mathtt{x}', \mathtt{x}'', \ldots$ as their respective inputs and outputs. On the level of synchronous programs, this corresponds to introduction of additional variables into the program: if a 1-channel is used in the implementation for communicating from $\mathtt{x}$ to $\mathtt{x}'$, where $\mathtt{x}'$ is an additional variable, additional identity equations $\mathtt{x}' := \mathtt{x};$ need to be added to the program, and references to $\mathtt{x}$ in the subprogram corresponding to the receiver thread of the channel need to be replaced by references to $\mathtt{x}'$.

We can trivially introduce an arbitrary number of additional variables and identity equations to a given SSDL program in PNF as follows: For a program $\mathtt{P}$, we denote its variable set as $X_{\mathtt{P}}$. According to the definition in Section 2.2.3, PNF-form SSDL programs consist of equations, for $l+1 \leq i \leq n$,

$$\bar{\mathtt{x}}_{Oi} \quad := \quad \mathtt{P}_i(\bar{\mathtt{x}}_{Ii}).$$

Now let $\bar{\mathtt{x}}_{Ii} = (\mathtt{x}_{Ii,1}, \ldots, \mathtt{x}_{Ii,j}, \ldots)$ be the input variable tuple for subpro-

gram $\mathtt{P_i}$. For any such $\mathtt{x}_{Ii,j}$, $\mathtt{P}$ can be modified to a program $\mathtt{P}[\mathtt{x}_{Ii,j}/\mathtt{x}'] \cup \{\mathtt{x}' := \mathtt{x}'_{Ii,j}\}$, where the variable *reference* $\mathtt{x}_{Ii,j}$ as a right-hand side parameter of subprogram $\mathtt{P_i}$ (and not any other refences to the same variable) is substituted by a reference to a new variable $\mathtt{x}'$, where $\mathtt{x}' \notin X_\mathtt{P}$, and the additional identity equation $\mathtt{x}' := \mathtt{x}_{Ii,j}$ is introduced to the program. Clearly, $X_{\mathtt{P}[\mathtt{x}_{Ii,j}/\mathtt{x}']} = X_\mathtt{P} \cup \{\mathtt{x}'\}$. This identity rewriting can be performed iteratively until the structure of the program matches the desired partition to 1-channels and threads. The program obtained will still be in PNF, where some subprograms $\mathtt{P_i}$ are the identity function, and will be referred to as a *linearization-matching* program. In a linearization-matching program, each thread in the linearization corresponds to a nonempty subset of variables in the program, and each 1-channel in the linearization corresponds to a pair of variables in the program.

Based on a linearization-matching SSDL program and its mapping to threads and 1-channels, we are ready to define the queuing capacity relation as follows.

Definition 4.1 ((Reduced) queuing capacity relation). *The reduced queuing capacity relation $\Delta \subseteq X \times \mathbb{N}_0 \times X$ is defined as the least relation such that, for all $\mathtt{x}, \mathtt{y} \in X$,*

$(\mathtt{x}, 1, \mathtt{y}) \in \Delta$ *if* $\mathtt{x}$ *and* $\mathtt{y}$ *are mapped to the same thread, or the same* 1*-channel.*

Clearly, Δ is a weighted ternary relation according to Def. 2.11. In the following, we shall make use of the minimum-weight reflexive-transitive closure of Δ, Δ^*, as defined in Def. 2.14. Δ^* then relates *all* directly or indirectly dependent pairs of variables, and yields the strongest synchronization constraint of all individual constraints combined. Being a uniquely weighted ternary relation, according to Def. 2.13, Δ^* can be written in functional notation as $\Delta^* : (X \times X) \to \mathbb{N}_0 \cup \{\infty\}$. Again, by intuition, $\Delta^*(\mathtt{x}, \mathtt{y}) = n$ incurs that, if purely implementation constraints are considered, $\mathtt{y}$'s symbol count may lead $\mathtt{x}$'s symbol count by at most n.

A given linearization has to satisfy both program and implementation constraints, and so either kind of constraint may dominate the other: For instance, in `twice-identity`, the general nature of a thread as an implementation mechanism does not impose a particular order between $\mathtt{w}$ and $\mathtt{x}$ emissions within one activation of the thread. The word $1^\mathtt{x}1^\mathtt{w}$ is therefore allowed by the structure of the *implementation* as a thread, $\mathtt{x}$'s symbol count may generally lead $\mathtt{w}$'s symbol count by one, and $\Delta^*(\mathtt{w}, \mathtt{x}) = 1$. However, the word $1^\mathtt{x}1^\mathtt{w}$ does not reflect the causality inherent in the *program*, and therefore violates the constraints given by $\xrightarrow{(.)}{}^*$, as the computation of $\mathtt{x}$'s value causally depends on the availability of $\mathtt{w}$'s value for a given

step. Thus, for the case of `x` and `w`, the program-induced constraint dominates over the implementation-induced constraint, so the word $1^{\mathtt{x}}1^{\mathtt{w}}$ will neither be accepted by this particular linearization of `twice-identity`, nor by any other linearization. We can capture this combination of variable synchronization relation and queuing capacity formally as the *joint synchronization map* $\bowtie$. We shall start out with a generic form of the joint synchronization map, which is parametrizable with a variable synchronization relation $\xrightarrow{(.)}{}^{*}_{X'}$ over variable subsets $X' \subseteq X$.

Definition 4.2 (Local joint synchronization map $\bowtie(.,.,.)$). *For a variable subset $X' \subseteq X$, let $\xrightarrow{(.)}{}^{*}_{X'} \subseteq X' \times \mathbb{N}_0 \times X'$ be a (local) variable synchronization relation over X'. The local joint synchronization map $\bowtie(.,.)$ maps a local variable synchronization relation $\xrightarrow{(.)}{}^{*}_{X'}$ and a variable pair $\mathtt{x}, \mathtt{y} \in X'$ to a synchronization count as follows: for all $\mathtt{x}, \mathtt{y} \in X'$, for all variable synchronization relations $\xrightarrow{(.)}{}^{*}_{X'}$,*

$$\bowtie(\xrightarrow{(.)}{}^{*}_{X'}, \mathtt{x}, \mathtt{y}) = \min\big(\xrightarrow{(.)}{}^{*}_{X'}(\mathtt{x}, \mathtt{y}),\ \Delta^*(\mathtt{x}, \mathtt{y})\big),$$

where for $n \in \mathbb{N}_0$, $\min(n, \infty) = \min(\infty, n) = n$ and $\min(\infty, \infty) = \infty$.

The local joint synchronization map hence models the combination of program constraints captured by $\xrightarrow{(.)}{}^{*}_{X'}$ for a possibly local variable subset $X' \subseteq X$, and implementation constraints in Δ^*. For most of the work on linearizations and property preservation in this chapter, we shall actually resort to a simplified version of $\bowtie$, which uses the global variable synchronization relation $\xrightarrow{(.)}{}^{*}$ over the entire variable set X of a given SSDL program:

Definition 4.3 ((Global) joint synchronization map $\bowtie(.,.)$). *For a given SSDL program, let $\xrightarrow{(.)}{}^{*}$ be the global variable synchronization relation over the variable set X as defined in Def. 2.10. The (global) joint synchronization map $\bowtie(.) : (X \times X) \to \mathbb{N}_0 \cup \{\infty\}$ is then defined as, for all $\mathtt{x}, \mathtt{y} \in X$,*

$$\bowtie(\mathtt{x}, \mathtt{y}) = \bowtie(\xrightarrow{(.)}{}^{*}, \mathtt{x}, \mathtt{y}).$$

Having defined the relationship between program-induced and implementation-induced synchronization on the level of variables, it is now time to take a closer look at the notion of words used in our framework, namely, synchronous words for describing the language of synchronous dataflow program, and linearized words for describing their linearized counterparts.

4.1.3 Words

In accordance with the more automata-theoretic setting of this chapter, we refer to sequences of symbols as words instead of streams. The core notation is essentially the same as introduced for streams in Section 2.2.2, and shall be briefly reviewed.

We write $\sigma \in \Sigma$ or $\gamma \in \Gamma$ for symbols, where Σ and Γ are used interchangeably to denote symbol sets. We typically use Σ to indicate symbols of synchronous words or symbols in general, and Γ for symbols of linearized words. $\Sigma^\omega = \Sigma^* \cup \Sigma^\infty$ is the set of all finite and infinite words over Σ. We typically use the letter w for synchronous words or words in the general case, and W for linearized words. Symmetric to the definitions for streams, we use the length operator $\#$, concatenation of words $w_1 \mathbin{.} w_2$, empty word ε, and projection operator $|_{(.)}$. Note that because of the properties of infinite words, the concatenation operator $\mathbin{.}$ is partial on $\Sigma^\omega \times \Sigma^\omega$: the concatenation of two words from Σ^ω is defined whenever the resulting word is in Σ^ω [DR95]. For a word $w \in \Sigma^\omega$, its i-th symbol is written as w_i, where $i > \#w \;\Rightarrow\; w_i = \varepsilon$. Similarly, the suffix of a word w starting at position i is written $w^{(i)} = w_i \mathbin{.} w_{i+1} \mathbin{.} \ldots$. A word w' is a prefix of another word w, written $w' \sqsubseteq w$, if there exists some (possibly empty) word w'' such that $w = w' \mathbin{.} w''$. For a word w, we write $\sqsubseteq^{-1}(w)$ to denote the set of all prefixes of w.

On the level of linearized words, we shall use symbol sets $\Gamma = (X \times V)$, where single symbols are written $\mathtt{a}^{\mathtt{x}}, \mathtt{b}^{\mathtt{y}}, \ldots$. The variable synchronization relation $\xrightarrow{(.)}{}^*$, the queuing capacity relation Δ^*, and the joint synchronization map $\bowtie$ over $X \times \mathbb{N}_0 \times X$ can be naturally extended to $(X \times V) \times \mathbb{N}_0 \times (X \times V)$: for all $\mathtt{x}, \mathtt{y} \in X$, it then holds that $\mathtt{x} R\, \mathtt{y} \;\Leftrightarrow\; \forall \mathtt{a}, \mathtt{b} \in V \,.\, \mathtt{a}^{\mathtt{x}} R\, \mathtt{b}^{\mathtt{y}}$, where R is one of $\left\{ \xrightarrow{(.)}{}^*, \Delta^* \right\}$, and $\bowtie$ is extended accordingly. For synchronous words, we introduce the notions of *variable partitions* and macrosymbols next.

Variable partitions and macrosymbols. A variable partition over a symbol set Γ with respect to a variable set $X = \{\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n\}$ partitions the symbol set into nonempty disjoint sets, $\Gamma = \Gamma_{\mathtt{x}_1} \cup \Gamma_{\mathtt{x}_2} \cup \cdots \cup \Gamma_{\mathtt{x}_n}$, where each symbol subset $\Gamma_{\mathtt{x}_i}$ is associated with variable $\mathtt{x}_i$, and $\mathtt{x}_i \neq \mathtt{x}_j \;\Leftrightarrow\; \Gamma_{\mathtt{x}_i} \cap \Gamma_{\mathtt{x}_j} = \emptyset$. For a word w with a variable partition for variable set X on Γ, we write $w|_{\mathtt{x}}$ as a shorthand for the projection $w|_{\Gamma_{\mathtt{x}}}$. Likewise, for some subset $X' \in X$, $X' = \{\mathtt{x}_i, \mathtt{x}_j, \ldots\}$, we write $w|_{X'}$ for $w|_{(\Gamma_{\mathtt{x}_i} \cup \Gamma_{\mathtt{x}_j} \cup \cdots)}$.

For synchronous words w, individual macrosymbols σ are taken from the set $\Sigma = (X \to V)$, as each macrosymbol constitutes a map from variables X to values V. For instance, program `twice-identity` from Fig. 4.2 is, on

the level of synchronous words, capable of producing macrosymbols of the form $\sigma = \{\mathtt{a}^{\mathtt{w}}\mathtt{a}^{\mathtt{x}}\mathtt{b}^{\mathtt{y}}\mathtt{b}^{\mathtt{z}}\}$, for some values $\mathtt{a}, \mathtt{b} \in V$, where clearly $\sigma \in (X \to V)$. Such macrosymbols $\sigma \in (X \to V)$ can again be projected to single variables $\mathtt{x}$: for $\sigma = \{\ldots \mathtt{a}^{\mathtt{x}} \ldots\}$, it simply holds that $\sigma|_{\mathtt{x}} = \{(\mathtt{x}, \sigma(\mathtt{x}))\} = \{\mathtt{a}^{\mathtt{x}}\}$. Projection can be naturally extended to variable sets $X' \subseteq X$, $X' = \{\mathtt{x_i}, \mathtt{x_j}, \ldots\}$ as $\sigma|_{X'} = \sigma|_{\mathtt{x_i}} \cup \sigma|_{\mathtt{x_j}} \cup \cdots$, and to $(X \to V)^*$-words as $\varepsilon|_{X'} = \varepsilon$ and $(w \centerdot \sigma)|_{X'} = w|_{X'} \centerdot \sigma|_{X'}$.

Macrosymbols can be partial, assigning values to subsets of variables $X' \subseteq X$. For two variable subsets $X' \subseteq X$, $X'' \subseteq X$, two macrosymbols $\sigma' \in (X' \to V)$ and $\sigma'' \in (X'' \to V)$ are *joinable*, written $\sigma' \sim \sigma''$, if for all $\mathtt{x} \in X' \cap X''$, it holds that $\sigma'(\mathtt{x}) = \sigma''(\mathtt{x})$. If two macrosymbols σ', σ'' are joinable, then their *join* is written $\sigma' ||| \sigma''$, and defined as $\sigma' ||| \sigma'' = \sigma' \cup \sigma''$. The join is in turn a macrosymbol in $((X' \cup X'') \to V)$. Joinability and joining can be naturally extended to pairs of words of the same length.

Synchronous words for synchronous stream tuples. As defined in Section 2.2.2, an SSDL program maps a tuple of (input) streams to a tuple of (output) streams. In the time-synchronous interpretation, the index position of a symbol in a stream corresponds to a global time index. Under this particular interpretation, it is quite natural to interpret a *tuple of sequences*, the input/output stream tuples given by the semantics, as a *sequence of tuples*, corresponding to a synchronous word $(X \to V)^\omega$.

Formally, for an SSDL program $\mathtt{P}$, we write a (partial) stream tuple as a (partial) function $X \to (V^\omega)$, where $X = X_I \cup X_L \cup X_O$ is the variable set of $\mathtt{P}$. The externally visible behavior of $\mathtt{P}$ is defined in Section 2.2.2 by a function $[\![\mathtt{P}]\!] : (X_I \to (V^\omega)) \to (X_O \to (V^\omega))$, mapping input stream tuples to output stream tuples. For an input stream tuple $\bar{x}_I \in X_I \to (V^\omega)$, we write $\min\#\bar{x}_I$ for the length of $\bar{x}_I$'s shortest component stream, where the length may be infinity. For a stream tuple $\bar{x}$, we write $(\bar{x})_{\min\#\bar{x}_I} \sqsubseteq \bar{x}$ for the prefix tuple of $\bar{x}$ such that each component of $(\bar{x})_{\min\#\bar{x}_I}$ is a prefix of the corresponding component stream of $\bar{x}$, and so that the length of each component prefix is $\min\#\bar{x}_I$.

Based on these preliminaries, the corresponding synchronous word language $L(\mathtt{P}) \subseteq (X \to V)^\omega$ for $\mathtt{P}$ is defined as the least set such that, for all $\bar{x}_I \in X_I \to (V^\omega)$, for all $\bar{x}_O \in X_O \to (V^\omega)$,

$$[\![\mathtt{P}]\!](\bar{x}_I) = \bar{x}_O \implies \big((\bar{x}_I)_{\min\#\bar{x}_I} ||| (\bar{x}_O)_{\min\#\bar{x}_I}\big) \in L(\mathtt{P})^1.$$

[1] We are a bit sloppy with notation, using stream tuples $\bar{x}$ to denote either a (partial) function to sequences $X \to (V^\omega)$, or the equivalent sequence of (partial) functions $(X \to V)^\omega$.

In the sequel, for the derivation of synchronous word languages and automata, we shall assume that in the PNF form of the SSDL program, the top-level program P's has no local variables in X_L, and hence all of its computations are visible in its synchronous word language. We can easily transform any PNF-form SSDL program into this form by changing all top-level `var` declarations to `output` declarations.

Linearizations of synchronous words. Based on the weighted variable dependency relation $\xrightarrow{(.)}$ defined in Def. 2.10, we can define a map lin from synchronous words $w \in (X \rightarrow V)^\omega$ to linearized words $W \in (X \times V)^\omega$. lin is parametrized with the joint variable synchronization relation $\bowtie$, as this relation has a constitutive influence on the possible interleavings in linearized words. We shall indicate this with a subscript as $lin_{\bowtie}$. Similar to the definition of $\bowtie$ in Defs. 4.2 and 4.3, we shall start out with a generic form of $lin_{\bowtie}$, which is parametrizable with a variable synchronization relation $\xrightarrow{(.)}{}^*_{X'}$ over variable subsets $X' \subseteq X$.

Definition 4.4 (Local linearization map $lin_{\bowtie}(.,.)$). *For a variable subset $X' \subseteq X$, let $\xrightarrow{(.)}{}^*_{X'} \subseteq X' \times \mathbb{N}_0 \times X'$ be a (local) variable synchronization relation over X'. Let $\bowtie(.,.,.)$ be the local joint synchronization map defined in Def. 4.2. We define a function $lin_{\bowtie}$ mapping $\xrightarrow{(.)}{}^*_{X'}$ and a synchronous word $w \in (X \rightarrow V)^\omega$ to sets of linearized words $\wp((X \times V)^\omega)$ as follows:*

For all variable synchronization relations $\xrightarrow{(.)}{}^*_{X'}$, for all $w \in (X' \rightarrow V)^\omega$, for all $W \in (X' \times V)^\omega$, it holds that $W \in lin_{\bowtie}(\xrightarrow{(.)}{}^*_{X'}, w)$ if and only if

$$\forall \mathtt{x},\mathtt{y} \in X. \left(\begin{array}{l} W|_{\mathtt{x}} = w|_{\mathtt{x}} \\ \wedge\ \forall W' \in \sqsubseteq^{-1}(W)\,.\ \left(\#W'|_{\mathtt{x}} \leq \#W'|_{\mathtt{y}} + \bowtie(\xrightarrow{(.)}{}^*_{X'}, \mathtt{x}, \mathtt{y}) \right) \end{array} \right).$$

Our definition of $lin_{\bowtie}$ in Def. 4.4 is based on the local joint synchronization relation $\bowtie(.,.,.)$ according to Def. 4.2, where a variable synchronization relation is passed as a first parameter. For most of the work on linearizations and property preservation in this chapter, we shall again use a simplified version of $lin_{\bowtie}$, which relies on the global variable synchronization relation $\xrightarrow{(.)}{}^*$ over the entire variable set X of a given SSDL program:

Definition 4.5 ((Global) linearization map $lin_{\bowtie}(.)$). *For a given SSDL program, let $\xrightarrow{(.)}{}^*$ be the global variable synchronization relation over the variable*

set X as defined in Def. 2.10. The (global) joint synchronization map $\bowtie(.) : (X \times X) \to \mathbb{N}_0 \cup \{\infty\}$ is then defined as, for all $w \in (X' \to V)^\omega$,

$$lin_{\bowtie}(w) = lin_{\bowtie}(\xrightarrow{(.)}{}^{*}, w).$$

For the linearization map, we show some properties. Lemma 4.1 shows, in a sense, that linearization has a different inherent flavor than "classical" behavioral refinement [Bro93][CGL94][DGG97][LGS+95]: each abstract behavior corresponds to a set of "refined" (linearized) behaviors, but the linearizations of different abstract behaviors never overlap. The relation between linearization and refinement-based design will be further discussed in Section 4.3. Property 4.2 is a derived statement about $(X \times V)^\omega$-equivalence classes. Lemmas 4.3 and 4.4 state that the concatenation of linearized words is a linearized word, and vice versa under certain conditions. Lemma C.1 in the appendix demonstrates that using the minimum-weight reflexive-transitive closure for the relations constituting $lin_{\bowtie}$ is not strictly necessary, and the usage of the reflexively-transitively reduced relations actually suffices from a formal standpoint.

Lemma 4.1 ($lin_{\bowtie}^{-1}$ is functional). *The following equivalent statements hold:*

1. *The inverse of $lin_{\bowtie}$, $lin_{\bowtie}^{-1}$, is a (partial) function from $(X \to V)^\omega$ to $(X \to V)^\omega$.*
2. *The respective $(X \times V)^\omega$-languages in $lin_{\bowtie}$'s image are disjoint.*
3. $\forall w, w' \in (X \to V)^\omega \,.\, \big(w \neq w' \iff lin_{\bowtie}(w) \cap lin_{\bowtie}(w') = \emptyset\big)$.

Proof. Clear from the constraint that, for all $\mathtt{x} \in X$, $W|_{\mathtt{x}} = w|_{\mathtt{x}}$: If two words $w, w' \in (X \to V)$ differ by at least one microsymbol $\mathtt{a}^{\mathtt{x}}$, any pair of words from $lin_{\bowtie}(w)$ and $lin_{\bowtie}(w')$ will also differ by that one microsymbol, and vice versa. □

Property 4.2 (Equivalence relation $\equiv_{lin}$ induced by $lin_{\bowtie}$). *Let Img_{lin} be the image of $(X \to V)^\omega$ through $lin_{\bowtie}$. Let $\equiv_{lin} \subseteq Img_{lin} \times Img_{lin}$ be the relation such that, for all $W, W' \in Img_{lin}$,*

$$W \equiv_{lin} W' \iff \exists \sigma \in (X \to V).W \in lin_{\bowtie}(\sigma) \wedge W' \in lin_{\bowtie}(\sigma).$$

Then $\equiv_{lin}$ is an equivalence relation over $Img_{lin} \times Img_{lin}$.

Definition 4.6 (n-synchronized variable set). *Let $X' \subseteq X$ be a variable set, and let $n \in \mathbb{N}_0$ be an integer. Then X' is n-synchronized in $\bowtie$, written $Sync_{\bowtie}^{n}(X)$, if for all variable pairs $\mathtt{x}, \mathtt{y} \in X'$, $\bowtie(\mathtt{x}, \mathtt{y}) \leq n$.*

Lemma 4.3 (Concatenation of linearized words is linearized word). *Let $w, w' \in (X \rightarrow V)^\omega$ be synchronous words such that $w \centerdot w'$ is defined. Then for all $W, W' \in (X \times V)^\omega$,*

$$W \in lin_{\bowtie}(w) \ \wedge \ W' \in lin_{\bowtie}(w') \implies (W \centerdot W') \in lin_{\bowtie}(w \centerdot w')$$

Lemma 4.4 (Linearized word with 1-synchronized variables is concatenation of linearized words). *Let X' be a variable set such that $Sync^1_{\bowtie}(X')$. Let $w, w' \in (X' \rightarrow V)^\omega$ be synchronous words such that $w \centerdot w'$ is defined. Then for all $W'' \in (X' \times V)^\omega$,*

$$W'' \in lin_{\bowtie}(w \centerdot w') \implies \exists W \in lin_{\bowtie}(w) \,.\, \exists W' \in lin_{\bowtie}(w') \,.\, W \centerdot W' = W''$$

Again, the proofs for both Lemmas 4.3 and 4.4 are found in the appendix.

Commutative diagram. Having defined the notion of linearized words, and having captured the relation between synchronous and linearized words in the linearization map $lin_{\bowtie}$, it should be interesting to compare our formalization with a different formal approach that has a more operational flavor, and bears a direct resemblance to the actual implementation structure in question. We shall introduce such a second approach: it will be based on synchronous automata representing SSDL programs, and linearized automata representing threads and 1-channels of the corresponding linearizations. The principal interrelation between the word-based and the automata-based formalization in our theory is illustrated by the commutative diagram in Fig. 4.6: Synchronous automata on the upper left, directly corresponding to SSDL programs, can be mapped to a combination of linearized automata, lower left, through the automata linearization map Lin, which shall be defined in the next section. The respective word languages are indicated on the right hand side. Clearly, the language of the linearized automata, $L(Lin(A_1) \| Lin(A_2) \| \cdots \| Lin(A_n))$, should coincide with the linearized language of the synchronous automata, $lin_{\bowtie}(L(A_1 \| A_2 \| \cdots \| A_n))$. The formal definition of synchronous and linearized automata is the subject of the next section. The equivalence of the language of linearized automata and the linearized language of synchronous automata will be the subject of Section 4.1.5.

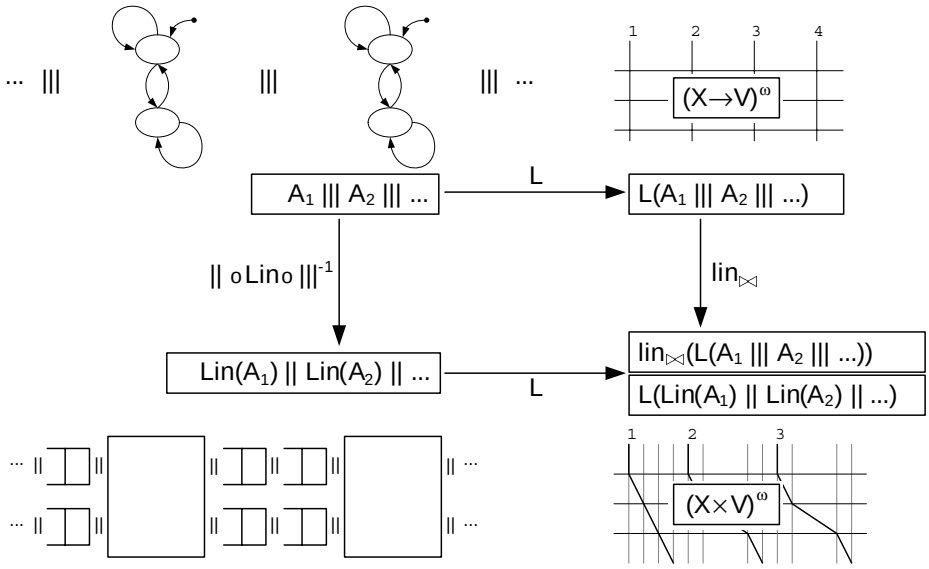


Figure 4.6: Commutative diagram for linearization maps Lin, $lin_{\bowtie}$

4.1.4 Automata

Synchronous automata. In the same way that synchronous words are an alternative model for stream tuples, synchronous automata A are an alternative model for the semantics of SSDL (sub)programs. For a partition of SSDL program P into subprograms $\mathtt{P}_1, \mathtt{P}_2, \ldots, \mathtt{P}_n$, the semantics of P can consequently be understood as the composition of synchronous automata $A_1 \| A_2 \| \ldots \| A_n$. For each subprogram $\mathtt{P}_i$, the word language of the subprogram $L(\mathtt{P}_i)$ is precisely the language $L(A_i)$. The direct synthesis of such automata from SSDL (sub)programs is standard [BFH+92]. For the reader unfamiliar with automaton synthesis from synchronous programs, the example at the end of this section will provide an intuitive understanding. We assume that automata A are finite, which typically corresponds to a finite value domain V for the SSDL program. Synchronous automata are also deterministic, corresponding to functionality of the SSDL semantics.

For a linearization-matching SSDL program P with subprograms $\mathtt{P}_1$, $\mathtt{P}_2$, $\ldots$, $\mathtt{P}_n$, variable set X, and value set V, a synchronous automaton A_i for subprogram $\mathtt{P}_i$ accepts macrosymbols over a subset of variables X, which corresponds to the inputs, local variables, and outputs of corresponding subprogram $\mathtt{P}_i$. We denote this subset as $X_{A_i} \subseteq X$. A_i accepts the language $L(A_i) = L(\mathtt{P}_i)$, where $L(A_i) \subseteq (X_{A_i} \to V)^\omega$.

A synchronous automaton A consists of an alphabet $\Sigma = (X_A \to V)$, states S, transition relation $\overset{(.)}{\underset{A}{\longmapsto}} \subseteq S \times (X_A \to V) \times S$, initial state $s_0 \in S$, and a weighted ternary relation $\overset{(.)}{\longrightarrow}{}^*_A \subseteq X_A \times \mathbb{N}_0 \times X_A$. An element $(s, \sigma, s') \in \overset{(.)}{\underset{A}{\longmapsto}}$ is written $s \overset{\sigma}{\underset{A}{\longmapsto}} s'$. When constructing an elementary synchronous automaton A from an SSDL subprogram, $\overset{(.)}{\longrightarrow}{}^*_A$ is initialized to the projection of the global variable synchronization relation $\overset{(.)}{\longrightarrow}{}^*$ (Def. 2.10) to $X_A \times \mathbb{N}_0 \times X_A$.

Synchronous composition $\|$ yields the product automaton, where transitions with joinable symbols are joined. More precisely, for two synchronous automata $A_1 = ((X_{A_1} \to V), S_1, \overset{(.)}{\underset{A_1}{\longmapsto}}, s_{01}, \overset{(.)}{\longrightarrow}{}^*_{A_1})$ and $A_2 = ((X_{A_2} \to V), S_2, \overset{(.)}{\underset{A_2}{\longmapsto}}, s_{02}, \overset{(.)}{\longrightarrow}{}^*_{A_2})$, the composed automaton is defined as

$$A_1 \| A_2 = \left(((X_{A_1} \cup X_{A_2}) \to V), S_1 \times S_2, \overset{(.)}{\underset{A_1 \| A_2}{\longmapsto}}, (s_{01}, s_{02}), \left(\overset{(.)}{\longrightarrow}{}^*_{A_1} \cup \overset{(.)}{\longrightarrow}{}^*_{A_2} \right)^* \right),$$

where $\left(\overset{(.)}{\longrightarrow}{}^*_{A_1} \cup \overset{(.)}{\longrightarrow}{}^*_{A_2} \right)^*$ denotes the minimum-weight reflexive-transitive

closure of the joined synchronization relations, as defined in Def. 2.14.

The composed transition relation $\overset{(.)}{\underset{A_1 \| A_2}{\longmapsto}}$ is defined as the least relation such that, for all $s_1, s_1' \in S_1$, $s_2, s_2' \in S_2$, $\sigma_1 \in (X_{A_1} \to V)$, $\sigma_2 \in (X_{A_2} \to V)$,

$$s_1 \overset{\sigma_1}{\underset{A_1}{\longmapsto}} s_1' \wedge s_2 \overset{\sigma_2}{\underset{A_1}{\longmapsto}} s_2' \wedge \sigma_1 \sim \sigma_2 \quad \Longrightarrow \quad (s_1, s_2) \overset{\sigma_1 \| \sigma_2}{\underset{A_1 \| A_2}{\longmapsto}} (s_1', s_2')$$

Clearly, synchronous composition $\|$ is commutative and associative.

For a synchronous automaton A, we can define a string-extended transition relation $\overset{(.)}{\underset{A}{\longmapsto}}^* \subseteq S \times (X_A \to V)^* \times S$ in the standard way as the least relation such that

$$s \overset{\varepsilon}{\underset{A}{\longmapsto}}^* s' \qquad \text{for all } s, s' \in S$$

$$\left(s \overset{w}{\underset{A}{\longmapsto}}^* s'' \wedge s'' \overset{\sigma}{\underset{A}{\longmapsto}} s'\right) \Longrightarrow s \overset{w \bullet \sigma}{\underset{A}{\longmapsto}}^* s' \qquad \text{for all } s, s', s'' \in S,\ w \in (X_A \to V)^*,\ \sigma \in (X_A \to V)$$

Synchronous automata accept both finite and infinite words, hence their language consists of a finite and infinite part, $L(A) = L^*(A) \cup L^\infty(A)$. All states of the automaton are by definition accepting: For finite words, this corresponds to prefix-closed languages. The finite language part $L^*(A)$ is thus defined as

$$L^*(A) = \left\{ w \in (X_A \to V)^* \mid \exists s \in S \,.\, s_0 \overset{w}{\underset{A}{\longmapsto}}^* s \right\}.$$

An infinite word $w \in L^\infty(A)$ is accepted if there exists an infinite state-transition sequence in A corresponding to w^∞. Alternatively, the infinite part $L^\infty(A)$ can be understood as the language formed by the upper bounds of infinite chains of prefixes in $L^*(A)$.

Linearized automata. Similar to synchronous automata being defined over synchronous $(X \to V)^\omega$ words, linearized automata are automata over linearized $(X \times V)^\omega$ words. A linearized automaton A with variable set $X_A \subseteq X$ is a tuple of alphabet $\Gamma = (X_A \times V)$, states S, transition relation $\overset{(.)}{\underset{A}{\longmapsto}} \subseteq S \times (X_A \times V) \times S$, initial state $s_0 \in S$, and accepting states $S_a \subseteq S$. Based on the transition relation $\overset{(.)}{\underset{A}{\longmapsto}}$, the string-extended transition relation $\overset{(.)}{\underset{A}{\longmapsto}}^*$ can be defined symmetrically to the construction for synchronous automata. In contrast to synchronous automata, composition $\|$ for linearized automata A_1, A_2 interleaves individual transitions,

and synchronizes on common symbols, in a way similar to process algebras [Hoa85][Mil80]. The composed automaton is defined as

$$A_1 \| A_2 = \big((X_{A_1} \times V) \cup (X_{A_2} \times V), S_1 \times S_2, \xmapsto[A_1 \| A_2]{(.)}, (s_{01}, s_{02}), S_{a1} \times S_{a2}\big).$$

The composed transition relation $\xmapsto[A_1 \| A_2]{(.)}$ is defined as the least relation such that, for all $s_1, s_1' \in S_1$, $s_2, s_2' \in S_2$, $\mathtt{x} \in X_{A_1} \cup X_{A_2}$, $\mathtt{a} \in V$,

$$\begin{aligned}
&\mathtt{x} \in X_{A_1} \setminus X_{A_2} \wedge s_1 \xmapsto[A_1]{\mathtt{a}^{\mathtt{x}}} s_1' && \implies (s_1, s_2) \xmapsto[A_1 \| A_2]{\mathtt{a}^{\mathtt{x}}} (s_1', s_2)\\
&\mathtt{x} \in X_{A_2} \setminus X_{A_1} \wedge s_2 \xmapsto[A_2]{\mathtt{a}^{\mathtt{x}}} s_2' && \implies (s_1, s_2) \xmapsto[A_1 \| A_2]{\mathtt{a}^{\mathtt{x}}} (s_1, s_2')\\
&\mathtt{x} \in X_{A_1} \cap X_{A_2} \wedge s_1 \xmapsto[A_1]{\mathtt{a}^{\mathtt{x}}} s_1' \wedge s_2 \xmapsto[A_2]{\mathtt{a}^{\mathtt{x}}} s_2' && \implies (s_1, s_2) \xmapsto[A_1 \| A_2]{\mathtt{a}^{\mathtt{x}}} (s_1', s_2')
\end{aligned}$$

Interleaving composition $\|$ is commutative and associative.

Linearized automata accept both finite and infinite words, where the set of accepting states is restricted to S_a. The finite language part $L^*(A)$ is then defined as

$$L^*(A) = \left\{ w \in (X_A \rightarrow V)^* \mid \exists s \in S_a \,.\, s_0 \xmapsto[A]{w}{}^{*} s \right\},$$

and an infinite word $w \in L^\infty(A)$ is accepted if the infinite state-transition sequence in A corresponding to w^∞ contains infinitely many states of the accepting set S_a.

Two linearized automata A_1 and A_2 may have the same languages, so that $L(A_1) = L(A_2)$. In this case, we write $A_1 \approx A_2$ to indicate that A_1 simulates A_2, and vice versa.

Thread automata. Thread automata are linearized automata with a given purpose: one thread automaton represents the family of sequential threads implementing a given synchronous automaton in our theory. We speak of a family of threads because for variables $\mathtt{x}, \mathtt{y} \in X$ with $\mathtt{x} \not\rightsquigarrow^* \mathtt{y}$, and values $\mathtt{a}, \mathtt{b} \in V$, a thread automaton for $\mathtt{x}, \mathtt{y}$ may accept both words $\mathtt{a}^{\mathtt{x}}\mathtt{b}^{\mathtt{y}}$ and $\mathtt{b}^{\mathtt{y}}\mathtt{a}^{\mathtt{x}}$, while for actually implemented threads, typically one of the two sequences must be chosen. Therefore, the automaton represents both the sequential thread producing $\mathtt{a}^{\mathtt{x}}\mathtt{b}^{\mathtt{y}}$ and the sequential thread producing $\mathtt{b}^{\mathtt{y}}\mathtt{a}^{\mathtt{x}}$.

For mapping synchronous automata to their thread automaton equivalent, we shall define a linearization map Lin. According to the intuition provided by Fig. 4.6, a thread automaton is synthesized such that it accepts precisely the linearized language of its corresponding synchronous automaton.

For a given synchronous automaton $A = \big(X_A \rightarrow V, S, \overset{(.)}{\underset{A}{\longmapsto}}, s_0, \overset{(.)}{\longrightarrow}_A^{\,*}\big)$, the corresponding thread automaton $Lin(A)$ has alphabet $\Gamma = (X_A \times V)$, states $Lin(S) = S \times (X_A \rightarrow V)$, transition relation $\overset{(.)}{\underset{Lin(A)}{\longmapsto}} \subseteq Lin(S) \times (X_A \times V) \times Lin(S)$, initial state $Lin(s_0) = (s_0, \emptyset)$, and accepting set $S_a = S \times \{\emptyset\}$. States $Lin(S)$ of the thread automaton $Lin(A)$ are pairs (s, σ), whereas $s \in S$ denotes a state of A, and $\sigma \in (X_A \rightarrow V)$ is a macrosymbol (partial over X_A) indicating the symbols that have already been accepted by the automaton since the last "empty" state $(s, \emptyset)$. This notion will be made clear in the construction below.

For the synchronous automaton A, define the local variable dependency order $\rightsquigarrow_A^*$ as

$$\rightsquigarrow_A^* \;=\; Unweight\big(\overset{(.)}{\longrightarrow}_A^{\,*}\big),$$

where the $Unweight$ operator is defined in Def. 2.12 in Section 2.2.4. The construction of the transition relation $\overset{(.)}{\underset{Lin(A)}{\longmapsto}}$ from the synchronous transition relation $\overset{(.)}{\underset{A}{\longmapsto}}$ and the local variable dependency order $\rightsquigarrow_A^*$ is achieved by applying the following rules in the construction:

1. For all $s, s' \in S$, $\sigma \in (X_A \rightarrow V)$, if $s \overset{\sigma}{\underset{A}{\longmapsto}} s'$, then unify states (s, σ) and $(s', \emptyset)$ in $Lin(S)$, and label the unified state as $(s', \emptyset)$.

2. For all $s, s' \in S$, $\sigma \in (X_A \rightarrow V)$ such that $s \overset{\sigma}{\underset{A}{\longmapsto}} s'$, for all partial macrosymbols $\sigma' \subset \sigma$ and $(X_A \times V)$-symbols $\gamma \in (\sigma \backslash \sigma')$ such that $\sigma' \subseteq \rightsquigarrow_A^{*-1}(\gamma)$,
$$(s, \sigma') \overset{\gamma}{\underset{Lin(A)}{\longmapsto}} (s, \sigma' \cup \{\gamma\}),$$

where $\rightsquigarrow_A^{*-1}(\gamma)$ is the (symbol-extended) inverse image of γ through the (symbol-extended) local variable dependency order $\rightsquigarrow_A^*$, as defined in Def. 2.9.

The latter rule ensures that for all $s, s' \in S$, and for all chains of microsymbols $\gamma_1, \gamma_2, \ldots, \gamma_n$ such that $(s, \emptyset) \overset{\gamma_1}{\underset{Lin(A)}{\longmapsto}} \overset{\gamma_2}{\underset{Lin(A)}{\longmapsto}} \cdots \overset{\gamma_n}{\underset{Lin(A)}{\longmapsto}} (s', \emptyset)$, the total order of the symbols in the chain corresponds to a linear extension of $\rightsquigarrow_A^*$.

1-channel automata. A 1-channel automaton is simply a thread automaton applied to an identity equation: while a generic thread automaton encodes the generic SSDL subprogram $\mathtt{P}_i$ referenced by an SSDL equation of

the form

$$\bar{\mathtt{x}}_{Oi} \quad := \quad \mathtt{P}_i(\bar{\mathtt{x}}_{Ii}),$$

the special case of a 1-channel automaton encodes the identity function corresponding to the equation, for some $\mathtt{x}, \mathtt{x}' \in X$,

$$\mathtt{x}' \quad := \quad \mathtt{x}.$$

Recall that identity equations are introduced to an SSDL program by rewriting it to the linearization-matching form. A 1-channel relates two variables, where one variable is the input, and the other is the output of the channel: each 1-channel in a chain adds a capacity of one symbol.

Because 1-channel automata always encode the same (identity) function, they have a canonical structure: For a 1-channel automaton encoding equation $\mathtt{x}' := \mathtt{x}$, denoted as $A_{\mathtt{x}' := \mathtt{x}}$, the following holds:

- The state set S is equal to $\{\{\mathtt{a}^{\mathtt{x}}\} \mid \mathtt{a} \in V\} \cup \{\emptyset\}$.
- The initial state s_0 is $\emptyset$.
- The set of acception states S_a is $\{\emptyset\}$.
- The transition relation $\xrightarrow[A_{\mathtt{x}' := \mathtt{x}}]{(.)}$ is the least relation such that

$$\emptyset \xrightarrow[A_{\mathtt{x}' := \mathtt{x}}]{\mathtt{a}^{\mathtt{x}}} \{\mathtt{a}^{\mathtt{x}}\} \quad \text{for all } \mathtt{a} \in V, \text{ and}$$

$$\{\mathtt{a}^{\mathtt{x}}\} \xrightarrow[A_{\mathtt{x}' := \mathtt{x}}]{\mathtt{a}^{\mathtt{x}'}} \emptyset \quad \text{for all } \mathtt{a} \in V.$$

Having thus defined our automata-based framework for formalizing realizations, we are ready to tackle another example of a linearized SSDL program, where we shall illustrate the consecutive steps in the linearization process.

Example (The frequency divider revisited.). We go back to the example of a frequency divider, `frequency-divide`, introduced in Sections 2.1.2 and 2.2.1. `frequency-divide` receives time ticks on input `in` and emits time ticks on output `out` on every second incoming tick. Tick variables may either have a value of `1` (tick present) or `0` (no tick present).

The SSDL program from Section 2.2.1 does not quite match the partitioning normal form (PNF) for an SSDL program. We

```
program frequency-divide-root;
  input  in;
  output out;

  program  frequency-divide;
    input    in;
    var      state, laststate;
    output   out;

    state     := if in=1 then 1-laststate else laststate fi;

    laststate := 1 fby state;
    out       := if in=1 then state else 0 fi;
  endprogram;

  out := frequency-divide(in);
endprogram;
```

Figure 4.7: Program `frequency-divide` in PNF

```
program frequency-divide-root;
  input  in;
  output in', out;

    ⋮


  in' := in;
  out := frequency-divide(in');
endprogram;
```

Figure 4.8: Program `frequency-divide` in linearization-matching PNF

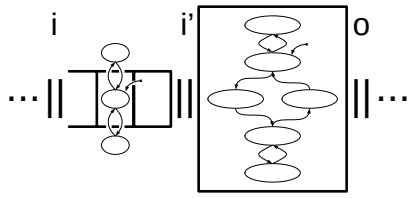
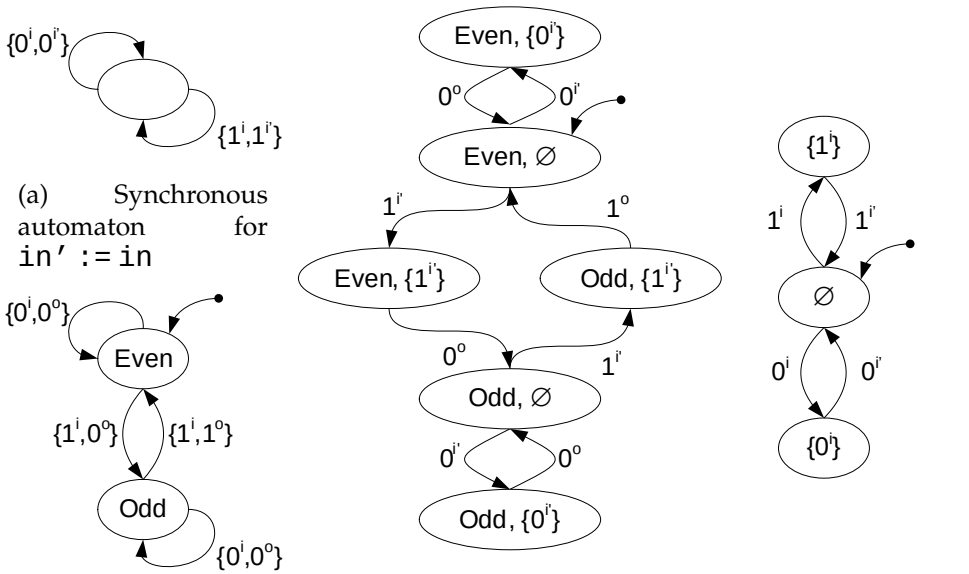


Figure 4.9: Composition of thread automaton, 1-channel automaton, and environment (schematic)

therefore surround `frequency-divide` with a superprogram, `frequency-divide-root`, shown in Fig. 4.8. We build a linearized implementation of `frequency-divide` with a channel for `in` of capacity 2. The capacity of 2 is realized by a 1-channel composed with the thread of `frequency-divide`, which provides capacity for one more `in` symbol, hence the total capacity for `in` is 2. As an output variable for the 1-channel and an input for the thread, an additional variable `in'` is introduced: Rewriting the program to its linearization-matching form, we end up with the program shown in Fig. 4.8. Figs. 4.10(a) and 4.10(b) show the corresponding synchronous automata for the identity equation `in' := in` and subprogram `frequency-divide`, respectively. Note that `in` is abbreviated `i` and `out` is written as `o`. In the automaton for `frequency-divide`, the two possible valuations for state variable `laststate` are mirrored by automaton locations named $Even$ and Odd for clarity.

Fig. 4.10(c) shows the linearized variant of the synchronous automaton in Fig. 4.10(b), which is simply the thread automaton for `frequency-divide`. Similarly, Fig. 4.10(d) shows the 1-channel automaton for `in` and `in`$'$. The thread automaton, the 1-channel automaton, and the environment of `frequency-divide`'s implementation are composed according to Fig. 4.9. The 1-channel automaton accepts `in` symbols as inputs, and `in`$'$ symbols as outputs, which are in turn accepted by the thread automaton.

With both an automata-based formalization, which closely resembles our intuition about the actual implemented system, and a word-based formalization, which concisely captures the nature of linearized words as a superimposition of synchronization constraints, we are thus ready to demonstrate the equivalence of both frameworks. This equivalence has already been hinted at in the commutative diagram of Fig. 4.6. Consequently, the next section will show that the language of the linearized automata indeed coincides with the linearized language of the synchronous automata.



(a) Synchronous automaton for `in' := in`

(b) Synchronous automaton for `frequency-divide`

(c) Thread automaton

(d) 1-channel automaton

Figure 4.10: Automata for `frequency-divide`

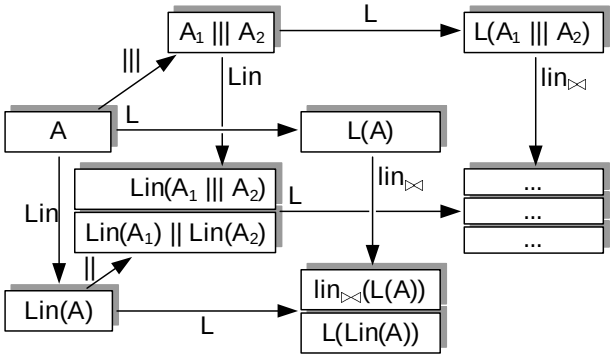


Figure 4.11: The commutative diagram refined

4.1.5 Equivalence

We go back to a refined version of our original commutative diagram in Fig. 4.6, shown in Fig. 4.11: The somewhat mysterious map $\| \circ Lin \circ |||^{-1}$ on the left-hand side of Fig. 4.6 is now more clearly illustrated. In order to show our original claim of interest, we will have to demonstrate that

$$lin_{\bowtie}(L(A_1|||A_2)) \;=\; L(Lin(A_1)\|Lin(A_2)) \left[=\; L(Lin(A_1|||A_2)) \right],$$

holds, as the case of more than two automata will then follow from commutativity and associativity of $|||$ and $\|$.

According to Fig. 4.11, it will thus be sufficient to concentrate on two central properties: Firstly, that the linearized language of a single synchronous automaton coincides with the language of its linearized counterpart. This is the property $lin_{\bowtie}(L(A)) = L(Lin(A))$ on the front lower right hand side of Fig. 4.11. Secondly, that construction of linearized automata from synchronous automata, Lin, distributes over composition. This is the property $Lin(A_1|||A_2) = Lin(A_1)\|Lin(A_2)$ on the back lower left hand side of Fig. 4.11. Our claim of interest will then follow from the combination of both properties.

The equivalence proof thus proceeds in several steps: Lemma 4.5 shows that acceptance of a macrosymbol σ by a synchronous automaton A follows from existential acceptance of $lin_{\bowtie}(\sigma)$ by its linearized automaton $Lin(A)$, and that universal acceptance of $lin_{\bowtie}(\sigma)$ by $Lin(A)$ follows from σ-acceptance by A. Lemma 4.6 states that linearized automata in a given state always accept sets of linearized words consistent with the partitions induced by $lin_{\bowtie}$, so that for given macrosymbol σ, existential and universal acceptance of words in $lin_{\bowtie}(\sigma)$ coincide. Lemmas 4.7 and 4.8 demonstrate that a synchronous automaton A in state s will accept a synchronous word

w and enter state s' precisely if its linearized counterpart $Lin(A)$ accepts linearized words $lin_{\bowtie}(w)$ on its way from state $(s, \emptyset)$ to $(s', \emptyset)$.

From Lemma 4.9, if follows that the linearized language of a single synchronous automaton coincides with the language of its linearized automaton. Lemma 4.10 shows that the linearization map Lin is compositional, so the linearized product of two linearized automata $Lin(A_1)$ and $Lin(A_2)$ simulates the linearized automaton of the synchronous product of their synchronous counterparts A_1 and A_2. Theorem 4.11, finally, states the main result of this section, coincidence of the language of linearized automata with the linearized language of the corresponding synchronous automata.

Lemma 4.5 (Existential and universal acceptance of $lin_{\bowtie}(\sigma)$-languages for $Lin(A)$). *Let A be a synchronous automaton, and let $Lin(A)$ be its linearized counterpart. For a given $\sigma \in (X_A \to V)$, we write $\exists W_\sigma$ short for $\exists W_\sigma \in lin_{\bowtie}(\xrightarrow{(.)}{}^*_A, \sigma)$, and symmetrically for $\forall W_\sigma$. Then the following holds, for all $\sigma \in (X_A \to V)$, for all $s, s' \in S$,*

$$1.\quad \exists W_\sigma.(s,\emptyset) \xmapsto[Lin(A)]{W_\sigma}{}^* (s',\emptyset) \implies s \xmapsto[A]{\sigma} s'$$
$$2.\quad \forall W_\sigma.(s,\emptyset) \xmapsto[Lin(A)]{W_\sigma}{}^* (s',\emptyset) \impliedby s \xmapsto[A]{\sigma} s'$$

Both statements can be deduced by a comparison of the construction of $Lin(A)$, based on $\xrightarrow{(.)}{}^*_A$, with $lin_{\bowtie}(\xrightarrow{(.)}{}^*_A, \sigma)$. The detailed proof can be found in the appendix.

Lemma 4.6 (Equivalence of existential and universal acceptance of $lin_{\bowtie}(w)$-languages for $Lin(A)$). *Let A be a synchronous automaton, and let $Lin(A)$ be its linearized counterpart. For a given $w \in (X_A \to V)$, we write $\exists W_w$ short for $\exists W_w \in lin_{\bowtie}(\xrightarrow{(.)}{}^*_A, w)$, and symmetrically for $\forall W_w$. Then the following holds, for all $\sigma \in (X_A \to V)$, for all $w \in (X_A \to V)^\omega$, for all $s, s' \in S$,*

$$1.\quad \exists W_\sigma.(s,\emptyset) \xmapsto[Lin(A)]{W_\sigma}{}^* (s',\emptyset) \iff \forall W_\sigma.(s,\emptyset) \xmapsto[Lin(A)]{W_\sigma}{}^* (s',\emptyset)$$
$$2.\quad \exists W_w.(s,\emptyset) \xmapsto[Lin(A)]{W_w}{}^* (s',\emptyset) \iff \forall W_w.(s,\emptyset) \xmapsto[Lin(A)]{W_w}{}^* (s',\emptyset)$$

We write $(s,\emptyset) \xmapsto[Lin(A)]{lin_{\bowtie}(\xrightarrow{(.)}{}^*_A, w)}{}^* (s',\emptyset)$ to denote both existential and universal acceptance of $lin_{\bowtie}(\xrightarrow{(.)}{}^*_A, w)$ by $Lin(A)$.

For the proof of statement 1., the $\Leftarrow$ direction is clear. The $\Rightarrow$ direction of statement 1. is an application of Lemma 4.5. Statement 2. follows from statement 1. by an inductive argument. Again, the detailed proof is given in the appendix.

Lemma 4.7 (Equivalence of σ-acceptance for A and $lin_{\bowtie}(\xrightarrow{(.)}{}^{*}_{A}, \sigma)$-acceptance for $Lin(A)$). *Let A be a synchronous automaton, and let $Lin(A)$ be its linearized counterpart. For all states $s, s' \in S$, and for all symbols $\sigma \in (X_A \to V)$,*

$$s \xmapsto[A]{\sigma} s' \iff (s, \emptyset) \xmapsto[Lin(A)]{lin_{\bowtie}(\xrightarrow{(.)}{}^{*}_{A}, \sigma)}{}^{*} (s', \emptyset)$$

Proof. Combine Lemma 4.5 and statement 1. of Lemma 4.6. □

Lemma 4.8 (Equivalence of w-acceptance for A and $lin_{\bowtie}(\xrightarrow{(.)}{}^{*}_{A}, w)$-acceptance for $Lin(A)$). *Let A be a synchronous automaton, and let $Lin(A)$ be its linearized counterpart. For all states $s, s' \in S$, and for all words $w \in (X_A \to V)^{\omega}$,*

$$s \xmapsto[A]{w}{}^{*} s' \iff (s, \emptyset) \xmapsto[Lin(A)]{lin_{\bowtie}(\xrightarrow{(.)}{}^{*}_{A}, w)}{}^{*} (s', \emptyset)$$

Proof. We write $Eq(s_1, w', s_2)$ as a shorthand for

$$s_1 \xmapsto[A]{w'}{}^{*} s_2 \iff (s_1, \emptyset) \xmapsto[Lin(A)]{lin_{\bowtie}(\xrightarrow{(.)}{}^{*}_{A}, w')}{}^{*} (s_2, \emptyset),$$

and show the property by induction over w:

1. $Eq(s, \varepsilon, s'')$ for all $s'' \in S$
2. $Eq(s, w, s'') \Longrightarrow Eq(s, w.\sigma, s''')$ for all $\sigma \in (X_A \to V)$, for all $s, s'', s''' \in S$

Statement 1. is clear by the construction rule for Lin: if $s \xmapsto[A]{\varepsilon}{}^{*} s'$, then $s = s'$, hence $(s, \emptyset) \xmapsto[Lin(A)]{lin_{\bowtie}(\xrightarrow{(.)}{}^{*}_{A}, \varepsilon)}{}^{*} s'$ for $lin_{\bowtie}(\xrightarrow{(.)}{}^{*}_{A}, \varepsilon) = \{\varepsilon\}$, and vice versa. Statement 2. follows from Lemma 4.7 combined with Lemmas 4.6 and 4.3. □

Lemma 4.9 (Equality of $lin_{\bowtie}\big(\xrightarrow{(.)}{}^{*}_{A}, L(A)\big)$ and $L(Lin(A))$). *Let A be a synchronous automaton. Then its linearized language is equal to the language of its linearized counterpart $Lin(A)$.*

$$lin_{\bowtie}\big(\xrightarrow{(.)}{}^{*}_{A}, L(A)\big) \;=\; L(Lin(A))$$

Proof. Direct application of Lemma 4.8, where s is set to the initial state s_0, and s' is any accepting state of A. □

Lemma 4.10 (Lin is compositional). *Let A_1, A_2 be synchronous automata. Then it holds that the linearized product of the linearized automata of A_1 and A_2 simulates the linearized automaton of the synchronous product of A_1 and A_2:*

$$Lin(A_1) \| Lin(A_2) \approx Lin(A_1 ||| A_2).$$

The proof for this lemma can be found in the appendix. It proceeds by establishing a bijection between states of $Lin(A_1)\|Lin(A_2)$ and states of $Lin(A_1|||A_2)$. To show equivalence of the two transition relations, it is demonstrated that the linearized composition $\|$ for transition paths individually ordered by $\rightsquigarrow^*_{A_1}$ and $\rightsquigarrow^*_{A_2}$ has the same effect as composing orders $\rightsquigarrow^*_{A_1}$ and $\rightsquigarrow^*_{A_2}$ in the synchronous composition $|||$, and then constructing linearized transition paths from the composite automaton through Lin.

Theorem 4.11 (Equality of $lin_{\bowtie}(L(A_1|||A_2||| \cdots |||A_n))$ and $L(Lin(A_1)\|Lin(A_2) \| \cdots \| Lin(A_n))$). *For some $n \in \mathbb{N}$, let $A_1, A_2, \ldots, A_n$ be synchronous automata corresponding to respective subprograms in a valid SSDL program in partioning normal form. Then it holds that*

$$lin_{\bowtie}(L(A_1|||A_2||| \cdots |||A_n)) \;=\; L(Lin(A_1)\|Lin(A_2)\| \cdots \| Lin(A_n)).$$

Proof. We note that for the composite synchronous automaton $A = A_1|||A_2||| \cdots |||A_n$, the variable synchronization relation $\xrightarrow{(.)}{}^{*}_{A}$ is equal to the global variable synchronization relation $\xrightarrow{(.)}{}^{*}$, as defined in Def. 2.10. The global relation $\xrightarrow{(.)}{}^{*}$, in turn, is used for constructing linearized words through $lin_{\bowtie}(.)$. The theorem then follows from commutativity and associativity of $|||$ and $\|$, and by application of Lemmas 4.9 and 4.10. □

After having spent the effort to formalize the notion of linearizations of synchronous programs, we shall put our linearization theory to work for one possible application: property preservation for Linear Temporal Logic (LTL) formulas. This will be the subject of the next section.

4.2 Property preservation

As we have laid out in Section 2.1, simple and formally defined models of computation such as synchronous languages provide an abstract way of specifying reactive systems and their behaviors. Analysis with respect to behavioral properties is typically much easier at this abstract level. Needless to say, the usefulness of the abstract verification result for practical applications hinges on the preservation of the property along the design process following the verification task, where for instance correct-by-construction synthesis may be used. This section shall examine the preservation issue for synchronous programs and their linearizations.

4.2.1 Overview

At the onset, it is not entirely clear exactly what kinds of properties are preserved through the process of linearization, as outlined in Section 4.1. Intuitively, for individual variables, the sequence of symbols observed in linearized runs will coincide with the sequence observed in runs of the synchronous program. Therefore, any assertion on the progression of an individual variable that holds for a synchronous program should also hold for its linearization. After briefly introducing temporal logic, we shall illustrate this with a small example.

For specifying abstract behavioral properties, we use the Linear Temporal Logic (LTL) framework [MP92]. LTL is a well-known formalism in the field of behavioral specification of reactive systems [Var01]. An LTL formula φ describes a set, or language, of infinite words; this set is also called the *model* of φ. An LTL formula over atomic propositions $\gamma \in \Gamma$ has the following syntax[2]:

$$\varphi ::= tt \mid \gamma \mid \neg\varphi \mid \varphi \vee \varphi' \mid \varphi \,\mathcal{U}\, \varphi',$$

where the LTL formula tt denotes the set of all words, the formula γ corresponds to the set of all words satisfying proposition γ, $\varphi \vee \varphi'$ is the set of all words satisfying φ or φ', and $\varphi \,\mathcal{U}\, \varphi'$ is the set of words for which φ holds at least until φ' holds (but φ' is eventually required). We write LTL_Γ for the set of LTL formulas over propositions Γ. This very basic set of operators already allows to express an extensive range of properties. Besides the standard derived operators such as $\wedge$ and $\Rightarrow$, three LTL-specific derived operators are the "eventually" operator $\Diamond$ defined as $\Diamond\varphi \;\Leftrightarrow\; tt \,\mathcal{U}\, \varphi$,

[2]We use the stuttering-free subset of LTL, therefore the next operator $\bigcirc$ is omitted.

the "always" operator $\Box$ defined as $\Box\varphi \Leftrightarrow \neg\Diamond(\neg\varphi)$, and the "weak until" operator $\mathcal{W}$ defined as $\varphi\,\mathcal{W}\,\psi \Leftrightarrow \varphi\mathcal{U}\psi \vee \Box\varphi$.

Example (Properties of frequency divider). We go back to the `frequency-divide` example from Section 4.1. A natural choice for atomic LTL propositions are symbols $\gamma \in (X{\times}V)$, where an LTL proposition $\gamma = \mathtt{a}^{\mathtt{x}}$ holds for a synchronous or linearized word if its first symbol for variable `x` is $\mathtt{a}^{\mathtt{x}}$. An example for a property for `frequency-divide` that can be specified in $LTL_{(X\times V)}$ is the following:

> "It always holds that an emission of $0^{\mathtt{out}}$ is eventually followed by an emission of $1^{\mathtt{out}}$, and that an emission of $1^{\mathtt{out}}$ is eventually followed by an emission of $0^{\mathtt{out}}$."

This property can be expressed in LTL:

$$\Box\big((0^{\mathtt{out}} \Rightarrow \Diamond 1^{\mathtt{out}}) \wedge (1^{\mathtt{out}} \Rightarrow \Diamond 0^{\mathtt{out}})\big)$$

Of course, the property will only hold if we make additional assumptions on the input `in`, but this is not in the focus of our demonstration. The only two atomic propositions in the formula, $0^{\mathtt{out}}$ and $1^{\mathtt{out}}$, are restricted to one variable, `out`. Consequently, both for synchronous or linearized words, it suffices to look at a projection of the word to `out` in order to judge whether the property holds. We make sure that the meaning of an atomic proposition $\mathtt{a}^{\mathtt{out}}$, for some $\mathtt{a} \in V$, is well-defined for the case of `out`-projected synchronous and linearized words: such a projection simply satisfies the proposition if its first symbol equals $\mathtt{a}^{\mathtt{out}}$.

Def. 4.5 asserts that, for all synchronous words $w \in (X \to V)^\infty$ and corresponding linearized words $W \in (X{\times}V)^\infty$, $W|_{\mathtt{out}} = w|_{\mathtt{out}}$, so the projection to `out` is equal for the synchronous and the linearized case. So we can easily conclude that if the above formula holds on all traces of `freq-divide`'s synchronous automaton, then it also holds on the projection to `out` of all linearized traces of `freq-divide`.

The example leaves open two important issues: How to deal with the frequent case of properties *relating several variables*? And how to define what it means for a *complete* linearized word to satisfy an LTL formula, as we have only talked informally about the *projection* of a linearized word to a single variable? These two open questions set the stage for the rest of this section.

4.2.2 LTL for synchronous and linearized words

LTL for synchronous words. Synchronous words are an instance of the more general class of *set-valued* infinite words: set-valued words have domain Σ^∞, where each $\sigma \in \Sigma$ constitutes a finite set $\{\gamma_1, \gamma_2, \ldots, \gamma_n\}$. We define the semantics of LTL over set-valued words for stuttering-free LTL formulas as follows: for all $w \in \Sigma^\infty$,

$$
\begin{array}{lcl}
w \models\!\!\models tt & & \\
w \models\!\!\models \gamma & \iff & \gamma \in w_1 \\
w \models\!\!\models \neg\varphi & \iff & w \not\models\!\!\models \varphi \\
w \models\!\!\models \varphi \vee \psi & \iff & (w \models\!\!\models \varphi) \vee (w \models\!\!\models \psi) \\
w \models\!\!\models \varphi \,\mathcal{U}\, \psi & \iff & \exists j \in \mathbb{N}.\big(w^{(j)} \models\!\!\models \psi \wedge \forall i \in \mathbb{N}.(i < j \Rightarrow w^{(i)} \models\!\!\models \varphi)\big).
\end{array}
$$

LTL for linearized words For linearized words, which are a special case of variable-partitioned words with symbol set $\Gamma = \Gamma_{x_1} \cup \Gamma_{x_2} \cup \cdots \Gamma_{x_n}$, a different interpretation of atomic propositions is needed: a proposition $\mathtt{a}^{\mathtt{x}}$ identifies those words where the first occurence of an $\Gamma_{\mathtt{x}}$-symbol is $\mathtt{a}^{\mathtt{x}}$. For variable-partitioned words Γ^∞, we thus define a linearized semantics for LTL formulas as follows: for all $W \in \Gamma^\infty$,

$$
\begin{array}{lcl}
W \models tt & & \\
W \models \mathtt{a}^{\mathtt{x}} & \iff & (W|_{\mathtt{x}})_1 = \mathtt{a}^{\mathtt{x}} \\
W \models \neg\varphi & \iff & W \not\models \varphi \\
W \models \varphi \vee \psi & \iff & (W \models \varphi) \vee (W \models \psi) \\
W \models \varphi \,\mathcal{U}\, \psi & \iff & \exists j \in \mathbb{N}.\big(w^{(j)} \models \psi \wedge \forall i \in \mathbb{N}.(i < j \Rightarrow w^{(i)} \models \varphi)\big).
\end{array}
$$

Our LTL interpretation is easily extended from single linearized words to sets of linearized words, which are the $(X \times V)^\infty$-languages. For a language $L \subseteq (X \times V)^\infty$, we write $L \models \varphi$ to denote that for all $W \in L$, it holds that $W \models \varphi$. Compared to the LTL semantics for set-valued words, we have simply adapted the semantics to the different interpretation of atomic propositions as $W \models \mathtt{a}^{\mathtt{x}} \Leftrightarrow (W|_{\mathtt{x}})_1 = \mathtt{a}^{\mathtt{x}}$. We ensure that for our nonstandard interpretation of atomic propositions, there exists a decision procedure for finite symbol sets Γ: any LTL formula under the linearized interpretation can be translated to an LTL formula under the (decidable) standard word interpretation $\models_W$ [SC85], where $W \models_W \gamma \Leftrightarrow W_1 = \gamma$, by exploiting the following equivalence:

$$
W \models \mathtt{a}^{\mathtt{x}} \iff W \models_W \Big(\bigvee_{\gamma \in (\Gamma \setminus \Gamma_{\mathtt{x}})} \gamma \Big) \,\mathcal{U}\, \mathtt{a}^{\mathtt{x}}.
$$

Note that the interpretation for atomic propositions $\mathtt{a}^{\mathtt{x}}$ is consistent with the equivalence classes induced by $\equiv_{lin}$ (Prop. 4.2) over $(X \times V)^\infty$-words. So if, for some $w \in (X \to V)^\infty$, any word in $lin_{\bowtie}(w)$ satisfies $\mathtt{a}^{\mathtt{x}}$, then all words in $lin_{\bowtie}(w)$ satisfy $\mathtt{a}^{\mathtt{x}}$. For general formulas $\varphi \in LTL_{(X \times V)}$, the linearized interpretation is *not* generally consistent with $\equiv_{lin}$: This is trivially clear for the formula $\neg \mathtt{1}^{\mathtt{x}} \,\mathcal{U}\, \mathtt{1}^{\mathtt{y}}$ applied to infinite synchronous words $w \in (\{\mathtt{x}, \mathtt{y}\} \to V)^\infty$ such that $w = \{\mathtt{1}^{\mathtt{x}}, \mathtt{1}^{\mathtt{y}}\}\{\ldots\}\ldots$ and $\mathtt{x} \not\rightsquigarrow^* \mathtt{y}$, so that the linearized words are of the form $lin_{\bowtie}(w) = \{\mathtt{1}^{\mathtt{x}}\mathtt{1}^{\mathtt{y}}\ldots,\ \mathtt{1}^{\mathtt{y}}\mathtt{1}^{\mathtt{x}}\ldots,\ \ldots\}$. While not having consistency for general $LTL_{(X \times V)}$-formulas, the best we can do is to establish a criterion which ensures consistency with $\equiv_{lin}$ at least for *some* formulas $\varphi \in LTL_{(X \times V)}$. In fact, consistency will later be shown to follow from another property, *preservability* for an LTL formula. A formula $\varphi \in LTL_{(X \times V)}$ is preservable if its $\models$-satisfaction for all linearizations of a synchronous word can be deduced from its $\models\!\!\models$-satisfaction for the synchronous word, and vice versa. Preservability is the criterion which does precisely address the two open issues of relating several variables and preserving complete linearized words, and shall be treated next.

4.2.3 Preservation

Preservable formulas. Having defined the LTL semantics for both synchronous and linearized words, we are now ready to examine the issue of property preservation. We can formally define what it means for an LTL formula over synchronous words to be preserved for the linearized words.

Definition 4.7 (Preservable formulas). *For a given variable set X, value set V, and linearization map $lin_{\bowtie}$, an LTL formula φ is called preservable, written $Pres(\varphi)$, if for all $w \in (X \to W)^\infty$, for all $W \in (X \times V)^\infty$,*

$$w \models\!\!\models \varphi \iff lin_{\bowtie}(w) \models \varphi.$$

Ideally, for an LTL formula φ, we should have a check on the structure of φ which rejects a formula if it is not preservable, and possibly accepts it otherwise. Such a check is described in the following: it recursively collects and returns the set of variables referenced in subformulas of φ, or returns a token indicating non-preservability of the subformula, resulting in non-preservability of the entire formula.

For implementing the check, we shall define a *preservable variable map* $PVar : LTL_{(X \times V)} \to \wp(X) \cup \{\top\}$ which maps a formula $\varphi \in LTL_{(X \times V)}$ to either the subset of those variables from X that are referenced by the atomic propositions in φ, or the special symbol $\top$, which indicates that the formula is not preserveable.

The codomain of $PVar$ is the powerset of X extended with top element $\top$, $\wp(X) \cup \{\top\}$. Over this domain, we define a partial order $\sqsubseteq$ such that $X' \sqsubseteq \top$ for all $X' \in \wp(X)$, and $X' \sqsubseteq X''$ if $X' \subseteq X''$ for all $X', X'' \in \wp(X)$. $\sqsubseteq$ induces a least upper bound operator $\sqcup$ in the usual way: we note that $X' \sqcup X'' = X' \cup X''$ for all $X', X'' \in \wp(X)$, and $X' \sqcup \top = \top \sqcup X' = \top$ for all $X' \in \wp(X) \cup \{\top\}$.

In the definition of $PVar$, we shall make use of the variable dependency relation $\rightsquigarrow^*$ from Def. 2.7. $\rightsquigarrow^*$ is naturally extended from a binary relation over X to a binary relation over the extended variable powerset $\wp(X) \cup \{\top\}$ as, for all $X', X'' \in \wp(X)$,

$$X' \rightsquigarrow^* X'' \iff \forall \mathtt{x}' \in X' \,.\, \forall \mathtt{x}'' \in X'' \,.\, \mathtt{x}' \rightsquigarrow^* \mathtt{x}'',$$

as well as $\top \not\rightsquigarrow^* X'$, $X' \not\rightsquigarrow^* \top$, and $\top \not\rightsquigarrow^* \top$. Likewise, the $Sync^n_{\bowtie}$ predicate of Def. 4.6 is extended to domain $\wp(X) \cup \{\top\}$ such that $\neg Sync^n_{\bowtie}(\top)$ for all $n \in \mathbb{N}_0$. With these preliminaries in place, we are ready to define the preservable variable map $PVar$ in Def. 4.8, as well as the auxiliary notion of a *single-variable conjunctive form* of an LTL formula in Def. 4.9. The single-variable conjunctive form is enforced for the first argument of an until-formula if the entire formula is to be preservable.

Definition 4.8 (Preservable variable map $PVar$). *Let $\varphi \in LTL_{(X \times V)}$ be an LTL formula. The preservable variable map $PVar$ is then defined over the structure of φ as follows:*

$$\begin{aligned}
PVar(tt) &= \emptyset \\
PVar(\mathtt{a}^{\mathtt{x}}) &= \{\mathtt{x}\} \\
PVar(\neg\varphi) &= PVar(\varphi) \\
PVar(\varphi \vee \psi) &= PVar(\varphi) \sqcup PVar(\psi) \\
PVar(\varphi \,\mathcal{U}\, \psi) &= \begin{cases} PVar(\varphi) \sqcup PVar(\psi) & \text{if } PVar(\psi) \rightsquigarrow^* PVar(\varphi) \\ & \text{and } \varphi \in \text{1-}CF \text{ and } Sync^1_{\bowtie}(PVar(\psi)) \\ \top & \text{otherwise} \end{cases}
\end{aligned}$$

Definition 4.9 (Single-variable conjunctive form). *We denote as single-variable formulas φ^1 those formulas for which $PVar(\varphi^1) \neq \top$ and $|PVar(\varphi^1)| \leq 1$, where $|PVar(\varphi^1)|$ denotes the cardinality of the variable set $PVar(\varphi^1)$. An LTL formula φ is in single-variable conjunctive form, written $\varphi \in$ 1-CF, if there exists an equivalent conjunctive formula $\varphi^{\text{1-}CF}$ of the form*

$$\varphi^{\text{1-}CF} ::= \varphi^1 \mid \varphi^{\text{1-}CF} \wedge \psi^{\text{1-}CF}$$

such that $\varphi^{\text{1-}CF} \iff \varphi$.

Having thus defined the preservable variable map $PVar$, we are ready to show a first property. It is a generalization of the intuitive claim made in the `frequency-divide` example at the beginning of the section: In order to assert an LTL property referring to variable `x` for a given word W, it suffices to examine its projection to variable `x`. Based on $PVar$, we generalize this claim to variable sets referenced by LTL formulas φ in Lemma 4.12.

Lemma 4.12 (Projection to $PVar(\varphi)$ preserves φ). *Let $\varphi \in LTL_{(X\times V)}$ be an LTL formula with $PVar(\varphi) \neq \top$. Then for all $W \in (X\times V)^\infty$ such that $W|_{PVar(\varphi)} \in (X\times V)^\infty$,*

$$W \models \varphi \iff W|_{PVar(\varphi)} \models \varphi.$$

Proof. By the definition of $\models$ for atomic propositions $\mathtt{a}^{\mathtt{x}}$, and by Def. 4.8, the linearized language $\{W \in (X\times V)^\infty \mid W \models \varphi\}$ is closed under introduction or removal of symbols not in $(PVar(\varphi)\times V)$. The property then follows. □

The following theorem demonstrates the central property of our preservation theory: Checking $PVar(\varphi)\neq\top$ for a given LTL formula φ is a sound check for preservability of φ, that is, the formula φ will always be rejected by the check if φ is not preservable.

Theorem 4.13 (Soundness of $PVar$). *Let $\varphi \in LTL_{(X\times V)}$ be an LTL formula. Then φ is preservable if $PVar(\varphi)\neq\top$:*

$$PVar(\varphi) \neq \top \implies Pres(\varphi).$$

The proof can be found in the appendix: it proceeds by structural induction over the constructs of LTL, whereby the $\mathcal{U}$ operator demands the most attention. This is reflected in the rather complex criterion defined within $PVar$ for $PVar(\varphi\ \mathcal{U}\ \psi) \neq \top$. With a sound criterion for checking preservability of LTL formulas in place, we can go back to the issue of $\equiv_{lin}$-consistency of formulas, which is now elegantly resolved.

Lemma 4.14 (Preservable formulas are $\equiv_{lin}$-consistent). *Let $\varphi \in LTL_{(X\times V)}$ be a preservable formula. Then φ is consistent with $\equiv_{lin}$, that is, for all $\equiv_{lin}$-equivalence classes $L \subseteq (X\times V)^\infty$,*

$$\exists W{\in}L\,.\,W\models\varphi \iff \forall W{\in}L\,.\,W\models\varphi.$$

Proof. $\Leftarrow$ is clear. For proving $\Rightarrow$, we note that as a consequence of Theorem 4.13 and Def. 4.8, $Pres(\varphi) \Rightarrow Pres(\neg\varphi)$. From $Pres(\neg\varphi)$, it follows that, for all $w \in (X \to V)^\infty$,

$$\neg(w \models\!\!\!\models \neg\varphi) \iff \neg(\forall W \in lin_{\bowtie}(w) \,.\, W \models \neg\varphi),$$

and hence,

$$w \models\!\!\!\models \varphi \iff \exists W \in lin_{\bowtie}(w) \,.\, W \models \varphi.$$

By definition of $\equiv_{lin}$, the equivalence class L is in the image of $lin_{\bowtie}$, so W is indeed in $lin_{\bowtie}(w)$ for some $w \in (X \to V)^\infty$, and the $\Rightarrow$ direction of the lemma thus follows from the above statement. □

We have thus shown that by computing $PVar$, we can identify LTL formulas that are preservable according to $Pres$. Our check has been shown to be sound, but not tight: indeed, there may be formulas for which $PVar(\varphi) = \top$, but which would actually be preserved according to $Pres$. However, we note that for the frequent case of variable subsets satisfying $Sync^1_{\bowtie}$, at least three important classes of properties will pass our check: *invariance* ($\Box$), *eventuality* ($\Diamond$), and *response*.

Property 4.15 (Preservation of $\Box$, $\Diamond$, response). *Let $\varphi, \psi \in LTL_{(X \times V)}$ be propositional LTL formulas, that is, combinations of atomic propositions tt and a^{x} (and derived propositions) involving only operators $\neg$ and $\vee$ (and derived operators). Then*

1. $Pres(\Box\varphi)$ *if* $Sync^1_{\bowtie}(PVar(\varphi))$
2. $Pres(\Diamond\varphi)$ *if* $Sync^1_{\bowtie}(PVar(\varphi))$
3. $Pres(\Box(\varphi \Rightarrow \Diamond\psi))$ *if* $Sync^1_{\bowtie}(PVar(\varphi) \sqcup PVar(\psi))$

Besides its actual methodical value, the work on property preservation has served yet another purpose: it has demonstrated that the framework of linearized words and automata from Section 4.1 is indeed a useful foundation for reasoning about implementations of synchronous dataflow programs.

4.3 Related Work

This section cites some related works both for the area of *linearizations of synchronous programs*, and *property preservation*.

Linearizations of synchronous programs. Approaches related to ours can be both found in the area of concurreny theory, and in the synchronous language community. Related to the former field of study, our framework uses equivalence classes over sets of linearized words for concurrent runs. This mechanism is equivalent to the framework of Mazurkiewicz traces [DR95], which shall be briefly discussed. Besides, two works from the synchronous language community on linearization of synchronous programs bear some resemblance to ours [CGP99][PBC05].

- Mazurkiewicz trace theory [DR95] is well-known for describing equivalence classes of concurrent runs, and can be seen as a generalization of the framework of linearized words, which applies only to the case of (variable-)partitioned sets of symbols. The concrete formalization of the equivalence relation is also different: while standard Mazurkiewicz trace theory uses symmetric dependency relations over symbols, and relates words in the same equivalence class through rewriting, our framework uses a dual criterion of equivalence, coincidence of projection $W_1|_{\mathtt{x}} = W_2|_{\mathtt{x}}$, and fulfillment of length constraints between prefixes $\#W|_{\mathtt{x}} \leq \#W|_{\mathtt{y}} + n$. An alternative formalization of the equivalence based on symmetric dependency relations over symbols and rewriting is possible for linearized words, but involves a more heavyweight formalization. For instance, the length constraint that for all prefixes W', $\#W'|_{\mathtt{x}} - 1 \leq \#W'|_{\mathtt{y}} \leq \#W'|_{\mathtt{x}} + 1$, is not directly expressible by dependency relations over symbols $(X \times V)$.

- [CGP99] and its formal underpinning in [CCGJ97] describe the distribution of automata in the imperative format OC [PBM+93]: OC, in turn, can be synthesized from synchronous programs. There are two main differences to our approach: Firstly, the primary program representation in [CCGJ97] is on the level of linearizations, so their framework does not directly address the question of mapping from synchronous programs/automata to linearized counterparts. In our framework, for instance, a direct relation between synchronous and linearized trace languages can be expressed. Secondly, on the level of linearized words, [CCGJ97] considers semi-commutations, instead

of symmetric dependency relations in our framework: it would also seem nontrivial to extend their chosen automata framework to the symmetric dependency case. Methodically, semi-commutations make the theory apply to less implementation-oriented scenarios: in a distribution theory for synchronous dataflow programs, semi-commutations capture program-induced causality only, assuming absence of implementation-induced synchronization, such as boundedness assumptions on channels, or mapping to common threads. Towards use in finite-state verification and preservation, it is well known that the class of regular languages is not closed under semi-commutation rewriting. Consequently, the connection to finite-state models, such as embodied by the LTL framework, is problematic for [CCGJ97], and would necessitate further constraints.

- [PBC05] describes a similar model to ours on the level of automata: like linearized automata, their "microstep automata" encode the possible causal behaviors of a synchronous program. In contrast to linearized automata, which model strictly length-preserving synchronous programs, microstep automata also encode non-length-preserving synchronous programs, and feature explicit clock transitions. The underlying purpose of the automata formalization is somewhat different in both frameworks: Microstep automata are embedded in a preservation theory for multiclock synchronous programs, where the focus is chiefly on semantics preservation under the assumption that absent messages are not communicated in an asynchronous implementation. Conversely, our linearization theory uses linearized automata to demonstrate the correctness, and provide an operational model for, the trace-based framework embodied by the linearization map $lin_{\bowtie}$. Relating trace languages by a linearization map has a more declarative flavor than automata construction, and is convenient for reasoning about relations between synchronous and linearized trace languages, as demonstrated by our LTL-based preservation theory.

Property preservation. There are two relevant areas for related research: firstly, direct research on *property preservation* for reactive systems, and secondly, research on temporal logics vs. concurrent traces.

- Most existing works on property preservation in the model checking community address the general paradigma of refinement-based design. Lemma 4.1 shows, in a sense, that linearization has a different methodical flavor than classical behavioral refinement either based

on trace sets [Bro93] or based on transition systems [CGL94][DGG97] [LGS+95]. Both in our approach and in refinement-based design, each abstract behavior corresponds to a set of "refined" (linearized) behaviors, but in linearization, "refinements" of different abstract behaviors never overlap. Furthermore, linearization theory is concerned with identifying admissible run-time observations in a concurrent systems, while refinement-based design typically aims at subsequently discarding possible behaviors until a more or less unique behavior is determined.

- The relationship between temporal logics and Mazurkiewicz traces has been explored e. g. in [Leu02]. The methodical background of concurrency-oriented temporal logics may be summarized as follows: a designer specifies a formula describing possible runs of a concurrent implementation. The concurrent implementation is hence the primary object of verification, and the focus is on the design of logics whose semantics is sufficiently fine-grained to distinguish relevant observations, yet is also consistent with the equivalence classes imposed by the underlying trace theory. In contrast, the methodical background of the preservation theory in Section 4.2 is to verify a property of a synchronous abstraction of a concurrent system, using the standard interpretation of LTL over set-valued words, and take advantage of a preservation principle for assessing properties of the system itself, using a closely related LTL interpretation. Overall, this leads to mutually different tradeoffs in the semantic interpretation of LTL formulas between the two approaches.

Chapter 5

Conclusion

5.1 Summary

In this thesis, we have motivated and introduced a simple synchronous dataflow language, which was used as the formal basis for the distribution-related work in Chapters 3 and 4. The concrete choice of computational model has been supported by a review of the particular features of dataflow languages, and by an evaluation of the synchronous dataflow model with respect to architecting and programming of software. Subsequently, the syntax and semantics of SSDL, a simple synchronous dataflow language, have been defined, including the particular aspect of causality analysis.

Based on the synchronous dataflow language SSDL, we have described two implementation schemes for synchronous dataflow programs: firstly, a singleprocessor implementation scheme, where a synchronous dataflow program is partitioned into several threads running on one processor, and the composite multithreaded program implements the semantics of the synchrous dataflow program using inter-thread communication primitives. Secondly, an implementation scheme for the multiprocessor case, where a synchronous dataflow program is split across several processors communicating over some event-triggered, bounded-jitter communication medium. Both implementation schemes have been shown to preserve certain critical aspects of the dataflow program's semantics.

As a theoretical foundation, we have formalized the notion of threads and finite channels in software-based implementations of synchronous dataflow program within the framework of linearized automata and linearized words. Together with transition systems and languages on the level of synchronous dataflow programs, this induces a dual refinement

relation between runs of synchronous programs and their linearizations, and between synchronous automata and their linearized counterparts. The two refinement maps for languages and automata were shown to be coincident. As an application of the theory, it was shown how a behavioral property verified on the abstract level can be used for inferring properties of the linearization.

Summarizing, with synchronous dataflow models as an example, the computer science community has arrived at specification formalisms and languages that allow expressing a system's behavior in a largely adequate and intuitive way, with sound formal foundations. For comprehension and validation, the advantages of using such formalisms over bottom-up design based on sequential language coding and low-level concurrency/communication handling are widely acknowledged. Deterministic models such as synchronous languages are especially attractive for reactive systems modeling, and can be efficiently implemented as real-time programs based on a variety of platform mechanisms for communication and synchronization, albeit under some idealizing assumptions.

5.2 Outlook

Based on recent results from this thesis and other works, one may be tempted to take as a given the big picture of a seamless development process for later phases of development, based exclusively on synchronous programming both at the architecture and programming level, combined with tightly defined refinement relations for obtaining real-world implementations. Despite advances in implementation techniques, one has to acknowledge that there are still some conceptual hurdles on the way towards this big picture.

The problem with determinism. One problem in correct-by construction implementation of synchronous programs are the strong, idealizing assumptions inherent in the communication and synchronization semantics, when taken at face value. As an example, synchronous dataflow programs describe *functional* programs, which map a timed input to a unique response. This inherently deterministic view on system construction leads to systems that may be expensive or impossible to implement on certain platforms. It is likely that one will not be able to hold up strict determinism in practice, especially in cost-sensitive domains such as Automotive.

This problem is nicely illustrated in the multiprocessor implementation scheme of Section 3.3, which needs to rely on a (lightweight) synchroniza-

tion mechanism in addition to the base protocol in order to implement the synchronous program in a fault-tolerant way, and needs to couple preservation guarantees with dedicated assumptions about operational conditions, as illustrated by Properties 3.22 and 3.23 in Section 3.3. It seems quite clear that the idealized operational conditions necessary for full preservation will not always hold in real-world settings.

As a side note, the problem of implementing strong assumptions on the level of models is not narrowly restricted to the field of synchronous languages: for a number of commercial approaches such as UML-RT [IBM03], if semantic preservation were taken seriously, similar problems would arise in correct-by-construction implementation. The problem is also not likely to vanish entirely with the (expensive) adoption of stronger base protocols [Kop97][Fle]. In fact, fault-tolerant *design*, which anticipates possible implementation failures at run-time in a correct-by-construction methodology, may remain just as relevant even with the advent of new platforms, as illustrated e. g. in [BBRN05].

We argue that for the specific problem of determinism and the general problem of model-based predictions about implementation behavior, there remains an unsettled terrain to be explored. Our argument is based on the observation that strict determinism is often not needed in an implementation, but is neverthess desirable on the level of models. For instance, we argue that not all messages in a synchronous program have the same character. On the one hand, embedded real-time control systems communicate sequences of sampled *state* values, often directly related to a measurement of their physical surroundings, where the precise preservation of the sampled sequence is of lesser importance than the preferrably short latency from measurement to reaction. In contrast, the approach as outlined in Chapter 4 is well-suited for *event* communication, where discrete events are communicated, and latency may be compromised in favor of preservation. This distinction is by no means unique to the embedded sector: Consider the two predominant transport protocols in the Internet, UDP for unreliable, connectionless, timely datagram traffic, and TCP for a reliable, connection-oriented service, with possible deferment of queued packets.

A broader view: soft conformance. As an outlook, we shall hint at a notion of "soft conformance", which is expected to formally grasp some of the inherent wisdom from the engineering field for building not quite deterministic, but perfectly correct systems with respect to the posed requirements. By providing a rigorous definition of such soft conformance

relations in the future, we may hope to transfer the field of complex real-time systems design from being more of an art to becoming a true engineering discipline.

We concentrate on concurrency and communication aspects of embedded systems design. With respect to these aspects, we envision a methodology that uses different degrees of determinism on different abstraction levels:

Loosely coupled, asynchronous models on the specification level The need for more loosely coupled notions of concurrency and communication is exemplified by the popularity of Message Sequence Charts (MSCs) [Int96] and related notations. These more example-oriented techniques support an explorative and loose style of design, which is often helpful when conceiving systems in early phases of development.

Tightly coupled, synchronous models on the design level. Exemplified by tools/formalisms such as AutoFOCUS, Simulink, SCADE, STATECHARTS, ESTEREL. Determinism on this level has the major advantage that (1) system-level validation and verification becomes possible (reduced state-space), (2) static guarantees about memory and time bounds are feasible, (3) efficient synthesis is available.

Loose concurrency and communication at the implementation level This is made necessary by (1) the desire to use cost-efficient hardware with good average-case utilization and (2) the complex and nondeterministic nature of platforms and combined HW/SW systems.

The relationship between the latter two levels is the subject of soft conformance: an implementation in soft conformance with a design if the implementation's behaviors are within an acceptable region around the original (design) behavior.

The difficulty is to define terms like "acceptable" and "region" satisfactorily. We will probably need to consider at least two complementary forms of relaxation

Discrete components/controllers, "message semantics" "Better late than never", at least up to some temporal/spatial bound, is the name of the game here. The design's traces are fully preserved (up to a bound), causal relations are respected, but the timing may be permuted. We believe that the formal relationship captured in Chapter 4 is a good starting point towards a notion of soft conformance in this direction.

(Sampled) continuous controllers, "state semantics" Here, one is typically rather interested in the "freshest value" with minimal delay, while discarding old samples if they arrive too late. Control loops are known to tolerate a bounded number of such losses, and sample loss may even be anticipated in the controller design.

This thesis is chiefly a contribution to the question of implementing message semantics for synchronous programs, and is therefore a first step in this direction. As a whole, soft conformance techniques for continuous and mixed discrete/controllers would be an important contribution for further improvements to the practical applicability of the synchronous approach.

Appendix A

Definitions and proofs for Chapter 2

Definition A.1 (Least upper bound (lub)). *For a partial order $(S, \sqsubseteq)$ and a subset $S' \subseteq S$, $s \in S$ is called an* upper bound *of S' iff*

$$\forall s' \in S' \,.\, s' \sqsubseteq s$$

s is called a least *upper bound of S', written $s = \bigsqcup S'$, iff*

1. *s is an upper bound of S', and*
2. *for all upper bounds s'' of S', $s \sqsubseteq s''$.*

Definition A.2 (Complete partial order (cpo)). *Let $(S, \sqsubseteq)$ be a partial order. By an ω-chain of the partial order, we denote an increasing chain $s_1 \sqsubseteq s_2 \sqsubseteq \cdots \sqsubseteq s_i \sqsubseteq \cdots$, where $s_i \in S$ are elements of the partial order.*

Then $(S, \sqsubseteq)$ is a complete *partial order iff any increasing chain $\{s_i \mid i \in \omega\}$ of elements $s_i \in S$ has a least upper bound $\bigsqcup\{s_i \mid i \in \omega\}$ in S.*

Definition A.3 (Pointwise extension of $\sqsubseteq$, $\bigsqcup$ to finite products). *Let S_1, S_2, ..., S_n be cpos with order $\sqsubseteq$ and lub $\bigsqcup$. The extension of $\sqsubseteq$ and $\bigsqcup$ to the finite product $S_1 \times S_2 \times \cdots \times S_n$ is defined as, for $s_i, s_i' \in S_i$, $S_i' \subseteq S$,*

$$(s_1, s_2, \ldots, s_n) \sqsubseteq (s_1', s_2', \ldots, s_n') \quad \textit{iff} \quad s_1 \sqsubseteq s_1' \wedge s_2 \sqsubseteq s_2' \wedge \cdots \wedge s_n \sqsubseteq s_n'$$

$$\bigsqcup(S_1' \times S_2' \times \cdots \times S_n') \;=\; \Big(\bigsqcup S_1', \bigsqcup S_2', \ldots, \bigsqcup S_n'\Big)$$

Property A.1 (Length-preservation for function tuples). *Let $f_1 : S \to S_1$, $f_2 : S \to S_2$, ..., $f_n : S \to S_1$ be length-preserving functions. Define the function*

$$\langle f_1, f_2, \ldots, f_n\rangle(s) = \big(f_1(s), f_2(s), \ldots, f_n(s)\big) \textit{ for } s \in S.$$

Then function $\langle f_1, f_2, \ldots, f_n\rangle$ is length-preserving.

Property A.2 (Length-preservation for function composition). *The composition of two length-preserving functions f and g, $f \circ g$, is length-preserving.*

Property A.3 (Continuity of function tuples). *Let $f_1 : S \to S_1$, $f_2 : S \to S_2$, ..., $f_n : S \to S_1$ be continuous functions. Define the function*

$$\langle f_1, f_2, \ldots, f_n \rangle(s) = \big(f_1(s), f_2(s), \ldots, f_n(s)\big) \text{ for } s \in S.$$

Then function $\langle f_1, f_2, \ldots, f_n \rangle$ is continuous.

Property A.4 (Continuity of function composition). *The composition of two continuous functions f and g, $f \circ g$, is continuous.*

Definition A.4 (Prefix closure). *For a stream domain V^ω, a subset $L \sqsubseteq V^\omega$ is prefix-closed iff*

$$x \in L \wedge x' \sqsubseteq x \Rightarrow x' \in T \text{ for all } x, x' \in V^\infty$$

Lemma A.5 (Length preservation and monotony implies prefix closure of image.). *Let $f : (V^*)^n \to (V^a st)$ be a length-preserving and monotonic function with mapping n-tuples of streams $\bar{x} \in (V^*)^n$ to streams $y \in V^*$. Then f's image on V^* is prefix-closed.*

Proof. Let $y = f(\bar{x})$ be any element of f's image. Let $y' \sqsubseteq y$ be any prefix of y. For $m = \min(\#\bar{x})$, let L denote the set

$$L = \{z_0 = f(\varepsilon^n), z_1 = f(\bar{x_1}), z_2 = f(\bar{x_1} \centerdot \bar{x_2}), \ldots, z_m = f(\bar{x}) = y\}.$$

We note that because of length-preservation, there exists a $\bar{k} \in \mathbb{N}_0^n$ such that for all stream tuples $\bar{x} \in (V^*)^n$, $\#f(\bar{x}) = \min(\#\bar{x} + \bar{k})$. Clearly, $\#(\varepsilon^n) = (0, 0, \ldots, 0)$, $\#\bar{x_1} = (1, 1, \ldots, 1)$, and so on, so there exists a unique $k \in \mathbb{N}_0$ such that for all z_i with $0 \leq i \leq n$, it holds that $\#z_i = i + k$, and $z_i \sqsubseteq z_{i+1}$ because of monotony. Z is obviously part of f's image. But y' must be equal to $z_{(\#y'-k)}$, because there can be only one prefix of y of length $\#y'$, and therefore $y' \in Z$. So for any y, any prefix y' is in f's image. Consequently, the image of f is prefix-closed. □

Lemma A.6 (Transitive and irreflexive relation). *For a set D, let R be some transitive and irreflexive relation over $D \times D$. Then R is antisymmetric.*

Proof. R contains only distinct pairs. For distinct $d, d' \in D$, assume there is a symmetric pair dRd', $d'Rd$. But then transitivity forces dRd and $d'Rd'$, leading to a contradiction as R is required to be irreflexive. □

Appendix B

Definitions and proofs for Chapter 3

Upper and Lower Bounds. We will use the following properties for working with upper and lower bounds in the proofs for Lemmas 3.14 and 3.17. Let S be a set, let $F, G : S \to \mathbb{R}$ be functions from S to the reals such that upper and lower bounds exist for F, G, and let $\min_{s \in S}(F(s))$ and $\max_{s \in S}(F(s))$ be the lower and upper bounds of F on S, respectively. Furthermore, let C be some constant. Then the following properties hold:

$$\max_{s \in S}(C) = C \tag{B.1}$$

$$\min_{s \in S}(C) = C \tag{B.2}$$

$$\max_{s \in S}(F(s) + G(s)) \leq \max_{s \in S}(F(s)) + \max_{s \in S}(G(s)) \tag{B.3}$$

$$\min_{s \in S}(F(s) + G(s)) \geq \min_{s \in S}(F(s)) + \min_{s \in S}(G(s)) \tag{B.4}$$

$$\max_{s \in S}(-F(s)) = -\min_{s \in S}(F(s)) \tag{B.5}$$

$$\min_{s \in S}(-F(s)) = -\max_{s \in S}(F(s)) \tag{B.6}$$

$$\max_{s \in S}(F(s)) < \min_{s \in S}(G(s)) \implies \forall s \in S.F(s) < G(s) \tag{B.7}$$

$$\max_{s \in S}(F(s)) \leq \min_{s \in S}(G(s)) \implies \forall s \in S.F(s) \leq G(s) \tag{B.8}$$

Lemma 3.3 (Harmonic clocks, properties). *Let T_i and T_j be two tasks with harmonic clocks $c_i \preceq c_j$. Then*

$$1. \quad C_j \circ Rel_i \circ C_i = C_j$$

$$2. \quad C_j \circ Rel_i^{\ominus} \circ (Id+1) \circ C_i = C_j$$

$$3. \quad Rel_i \circ C_i \circ Rel_j = Rel_j$$

Proof. For 1., Prop. 3.2 yields, for $c_i \preceq c_j$,

$$\begin{array}{rcll} Rel_i \circ C_i & \geq & Rel_j \circ C_j & \\ C_j \circ Rel_i \circ C_i & \geq & C_j \circ Rel_j \circ C_j & \text{(monotony of } C_j\text{)} \\ C_j \circ Rel_i \circ C_i & \geq & C_j & (C_j \circ Rel_j = Id) \end{array}$$

But also, due to $Rel_i \circ C_i \leq Id$, $C_j \circ Rel_i \circ C_i \leq C_j$, it holds that $C_j \circ Rel_i \circ C_i = C_j$.

For 2., we start with the second inequation of Prop. 3.2:

$$\begin{array}{rcll} Rel_i^{\ominus} \circ (Id{+}1) \circ C_i & \leq & Rel_j^{\ominus} \circ (Id{+}1) \circ C_j & \\ C_j \circ Rel_i^{\ominus} \circ (Id{+}1) \circ C_i & \leq & C_j \circ Rel_j^{\ominus} \circ (Id{+}1) \circ C_j & \text{(monotony of } C_j\text{)} \\ C_j \circ Rel_i^{\ominus} \circ (Id{+}1) \circ C_i & \leq & C_j & (C_j \circ Rel_j^{\ominus} = (Id{-}1)) \end{array}$$

But also $Rel_i^{\ominus} \circ (Id{+}1) \circ C_i \geq Id$, hence $C_j \circ Rel_i^{\ominus} \circ (Id{+}1) \circ C_i \geq C_j$, and the equality follows.

For 3., Prop. 3.2 yields, again,

$$Rel_j \circ C_j \leq Rel_i \circ C_i.$$

Chaining with Rel_j and simplifying $C_j \circ Rel_j = Id$, we obtain

$$Rel_j \leq Rel_i \circ C_i \circ Rel_j.$$

We note that also $Id \geq Rel_i \circ C_i$, hence $Rel_j \geq Rel_i \circ C_i \circ Rel_j$, so the equality in 3. follows. □

Lemma 3.4 (Harmonic clocks, write sampling). *Let T_i and T_j be two tasks with harmonic clocks $c_i \preceq c_j$, and let task T_i be write sampling with respect to task T_j. Then the following holds:*

1. $Rel_j \circ C_j \quad < \quad W_i \circ C_i$
2. $C_i \circ Rel_j \circ C_j \quad = \quad C_i \circ W_i \circ C_i.$

And if $c_i \succeq c_j$, the following holds:

3. $C_i \circ W_i \quad = \quad Id.$

Proof. For part 1., we know from Def. 3.3

$$Rel_i \circ C_i \circ Rel_j \circ C_j < W_i \circ C_i < Rel_i \circ (Id{+}1) \circ C_i \circ Rel_j \circ C_j.$$

From Prop. 3.2, it follows that

$$Rel_j \circ C_j \quad \leq \quad Rel_i \circ C_i$$

and therefore, chaining with the monotonic function $Rel_j \circ C_j$ and simplifying $C_j \circ Rel_j = Id$,

$$Rel_j \circ C_j \leq Rel_i \circ C_i \circ Rel_j \circ C_j.$$

Combining with the left-hand side inequation of Def. 3.3 yields

$$Rel_j \circ C_j < W_i \circ C_i,$$

which proves part 1. of the lemma.

For part 2., we expand Def. 3.3 with $Rel_j = Rel_i \circ C_i \circ Rel_j$ from Lemma 3.3, so

$$Rel_i \circ C_i \circ Rel_j \circ C_j < W_i \circ C_i < Rel_i \circ (Id{+}1) \circ C_i \circ Rel_j \circ C_j.$$

Using the left limit of $Rel_j = Rel_i \circ C_i \circ Rel_j$, this can be rewritten as

$$Rel_i \circ C_i \circ Rel_j \circ C_j < W_i \circ C_i \leq Rel_i^{\ominus} \circ (Id{+}1) \circ C_i \circ Rel_j \circ C_j.$$

Chaining with the monotonic function C_i yields

$$C_i \circ Rel_i \circ C_i \circ Rel_j \circ C_j \leq C_i \circ W_i \circ C_i \leq C_i \circ Rel_i^{\ominus} \circ (Id{+}1) \circ C_i \circ Rel_j \circ C_j.$$

Now using $C_i \circ Rel_i = Id$ and $C_i \circ Rel_i^{\ominus} = Id{-}1$, we can simplify to

$$C_i \circ Rel_j \circ C_j \leq C_i \circ W_i \circ C_i \leq (Id{-}1) \circ (Id{+}1) \circ C_i \circ Rel_j \circ C_j,$$

which proves the equality of part 2.

For part 3., we use $C_i \circ Rel_j \circ C_j = C_i$ from Lemma 3.3 to simplify Def. 3.3:

$$Rel_i \circ C_i < W_i \circ C_i < Rel_i \circ (Id{+}1) \circ C_i.$$

Eliminating C_i, and replacing strict inequality with the a limit, It follows that

$$Rel_i < W_i \leq Rel_i^{\ominus} \circ (Id{+}1).$$

Chaining with the monotonic function C_i, and using $C_i \circ Rel_i = Id$, $C_i \circ Rel_i^{\ominus} = Id{-}1$ yields

$$Id \leq C_i \circ W_i \leq (Id{-}1) \circ (Id{+}1)$$

which proves the property. □

Lemma 3.14. *Let N be some non-master node in a cascade under normal operating conditions. Then $(j-1)$-synchronization of N implies j-receptiveness of N.*

Proof. For j-receptiveness of N, Equations 3.5 (case (1)) and 3.6 (case (2)) must hold.

(1) We can rewrite Equation 3.5 as

$$Q + (t_{i,j-1} - t_{0,j-1}) < d_{\min}(0,i) + (t_{0,j} - t_{0,j-1}).$$

This condition is quantified over all possible executions of the cascade under normal operating conditions: we indicate the set of such executions with NOC. Using Equations B.1–B.8, we eliminate the quantification and use lower/upper bounds instead:

$$Q + \max_{(NOC)}(t_{i,j-1} - t_{0,j-1}) < d_{\min}(0,i) + \min_{(NOC)}(t_{0,j} - t_{0,j-1}).$$

From Section 3.3.3, it follows that $T/(1+\varepsilon)$ is a lower bound for $t_{0,j} - t_{0,j-1}$. Because N is $(j-1)$-synchronized by assumption, the bounded message jitter property from Section 3.3.3 yields $\max_{(NOC)}(t_{i,j-1} - t_{0,j-1}) = d_{\max}(0,i)$. Substituting and using Equation 3.1, the following must hold:

$$Q + \textstyle\sum_{s_j \in Li(N_i)} \Delta_{li}(s_j) < \dfrac{T}{1+\varepsilon}.$$

Using the $(T-Q)/2$ bound for the message jitter from Definition 3.5, we have to show

$$Q + \frac{T-Q}{2} < \frac{T}{1+\varepsilon}.$$

Solving for Q yields the condition ($0 \leq \varepsilon < 1$)

$$Q < T\left(\frac{1-\varepsilon}{1+\varepsilon}\right),$$

which holds for $Q < T \cdot (1-2\varepsilon), 0 \leq \varepsilon < 1$. This proves case 1.

(2) By assumption, it is true that $(j-1)$-synchronized(N). Rewriting Equation 3.6 and using upper/lower bounds yields:

$$\frac{T_{ma}}{1+\varepsilon} + \min_{(NOC)}(t_{i,j-1} - t_{0,j-1}) > \max_{(NOC)}(t_{0,j} - t_{0,j-1}) + d_{\max}(0,i)$$

With $T/(1-\varepsilon)$ as an upper bound for $t_{0,j} - t_{0,j-1}$, $d_{\min}(0,i)$ as a lower bound for $(t_{i,j-1} - t_{0,j-1})$, and Equation 3.1, we obtain:

$$\frac{T_{ma}}{1+\varepsilon} > \frac{T}{1-\varepsilon} + \textstyle\sum_{s_j \in Li(N_i)} \Delta_{li}(s_j).$$

Substituting our choice for T_{ma} from equation 3.2 and solving for $\sum_{s_j \in Li(N_i)} \Delta_{li}(s_j)$ with $0 \leq \varepsilon < 1$ yields:

$$\sum_{s_j \in Li(N_i)} \Delta_{li}(s_j) < \frac{T(1-5\varepsilon)}{2(1-\varepsilon^2)}.$$

For $0 \leq \varepsilon < 1$, this inequation follows from Definition 3.5. This proves case 2. □

Lemma 3.17. *Let N be some non-master node in a cascade under transient fault conditions. Then if there exists an n, $1 \leq n \leq n_{pf}$, such that N is $(j-n)$-synchronized, then N is j-receptive.*

Proof. By assumption, there exists an $n \in \{1, \ldots, n_{pf}\}$ such that N is $(j-n)$-synchronized. Let n' be the smallest such n. Then for j-receptiveness of N, Equations 3.5 (case (1)) and 3.6 (case (2)) must hold.

(1) We distinguish cases (1a) (N is $(j-1)$-synchronized) and (1b) (N is $(j-n')$-synchronized, and $2 \leq n' \leq n_{pf}$).

(1a) See case (1) of the proof for Lemma 3.14.

(1b) We can rewrite Equation 3.5 as

$$Q + (t_{i,j-1} - t_{0,j-1}) < d_{\min}(0,i) + (t_{0,j} - t_{0,j-1}).$$

The equation is implicitly quantified over the set of executions under transient fault conditions, TFC. Quantification is removed by taking upper/lower bounds:

$$Q + \max_{(TFC)}(t_{i,j-1} - t_{0,j-1}) < d_{\min}(0,i) + \min_{(TFC)}(t_{0,j} - t_{0,j-1}). \tag{B.9}$$

For obtaining an upper bound for $t_{i,j-1} - t_{0,j-1}$, we split the term $t_{i,j-1} - t_{0,j-1}$ using the identity

$$\begin{aligned} t_{i,j-1} - t_{0,j-1} \quad = \quad & (t_{i,j-n'} - t_{0,j-n'}) \\ & +(t_{i,j-1} - t_{i,j-n'}) \\ & -(t_{0,j-1} - t_{0,j-n'}). \end{aligned}$$

Taking the maximum over executions TFC, and using Equations B.1–B.6, we obtain

$$\begin{aligned} \max_{(TFC)}(t_{i,j-1} - t_{0,j-1}) \quad \leq \quad & \max_{(TFC)}(t_{i,j-n'} - t_{0,j-n'}) \\ & + \max_{(TFC)}(t_{i,j-1} - t_{i,j-n'}) \\ & - \min_{(TFC)}(t_{0,j-1} - t_{0,j-n'}), \end{aligned}$$

which can be resolved as follows:

- $\max_{(TFC)}(t_{i,j-n'} - t_{0,j-n'})$:
  We observe that, by assumption of Lemma 3.17 and using the right-hand side condition of Equation 3.4, $t_{i,j-n'} - t_{0,j-n'} \leq d_{\max}(0,i)$. Therefore, $d_{\max}(0,i)$ is a valid upper bound.

- $\max_{(TFC)}(t_{i,j-1} - t_{i,j-n'})$:
  According to the operational definition of N, N will first detect a message absence (yielding $T_{ma}/(1-\varepsilon)$ as an upper bound for the duration of cycle $j-n'$) and then perform $n'-2$ unsynchronized steps (yielding an upper bound of $(n'-2)\cdot T/(1-\varepsilon)$ for the remaining cycles). The total upper bound is $T_{ma}/(1-\varepsilon) + (n'-2)\cdot T/(1-\varepsilon)$.

- $\min_{(TFC)}(t_{0,j-1} - t_{0,j-n'})$:
  The lower bound for the duration of $n'-1$ cycles of the master is $(n'-1)\cdot T/(1+\varepsilon)$.

We substitute the upper bound for $t_{i,j-1} - t_{0,j-1}$ into Equation B.9, and use $T/(1+\varepsilon)$ as a lower bound for $t_{0,j} - t_{0,j-1}$ and Equation 3.1 for $d_{\max} - d_{\min}$. Then the following property remains to be shown:

$$\frac{T_{ma}}{1-\varepsilon} + (n-2)\frac{T}{1-\varepsilon} + Q + \textstyle\sum_{s_j \in Li(N_i)} \Delta_{li}(s_j) < n'\frac{T}{1+\varepsilon}.$$

Solving for n' results in the condition ($0 \leq \varepsilon < 1, T > 0$):

$$n' < \frac{1}{4T\varepsilon}\left(T(1+\varepsilon) - \left(2Q + 2\textstyle\sum_{s_j \in Li(N_i)} \Delta_{li}(s_j)\right)(1-\varepsilon^2)\right).$$

For $0 \leq \varepsilon < 1$, this holds if

$$n' < \frac{1}{4T\varepsilon}\left(T - \left(2Q + 2\textstyle\sum_{s_j \in Li(N_i)} \Delta_{li}(s_j)\right)\right).$$

This follows from $n' \leq n_{pf}$ and Equation 3.3, so we're done for case 1.

(2) We distinguish cases (2a) (N is $(j-1)$-synchronized) and (2b) (N is $(j-n')$-synchronized and $2 \leq n' \leq n_{pf}$).

(2a) See case (2) of the proof for Lemma 3.14.

(2b) For this case, it is true that $\neg(j-1)$-synchronized(N). Rewriting Equation 3.7 and using upper/lower bounds yields:

$$\frac{T}{1+\varepsilon} + \min_{(TFC)}(t_{i,j-1} - t_{0,j-1}) > \max_{(TFC)}(t_{0,j} - t_{0,j-1}) + d_{\max}(0,i). \tag{B.10}$$

A lower bound for $t_{i,j-1} - t_{0,j-1}$ is again found by splitting up the term and using Equations B.1–B.6:

- $\min_{(TFC)}(t_{i,j-n'} - t_{0,j-n'})$:
  By assumption of Lemma 3.17 and using the left-hand side condition of Equation 3.4, $t_{i,j-n'} - t_{0,j-n'} \geq d_{\min}(0,i)$, so $d_{\min}(0,i)$ is a valid lower bound.

- $\min_{(TFC)}(t_{i,j-1} - t_{i,j-n'})$:
  N will first detect a message absence (lower bound $T_{ma}/(1+\varepsilon)$) and then perform $n'-2$ unsynchronized steps (lower bound $(n'-2) \cdot T/(1+\varepsilon)$ for the remaining cycles). The total lower bound is $T_{ma}/(1+\varepsilon) + (n-2) \cdot T/(1+\varepsilon)$.

- $\max_{(TFC)}(t_{0,j-1} - t_{0,j-n'})$:
  The upper bound for the duration of $n'-1$ cycles of the master is $(n'-1) \cdot T/(1-\varepsilon)$.

With $T/(1-\varepsilon)$ as an upper bound for $t_{0,j} - t_{0,j-1}$ and Equation 3.1, substituting the above bounds into Equation B.10 yields:

$$\frac{T_{ma}}{1+\varepsilon} + (n'-1)\frac{T}{1+\varepsilon} > n'\frac{T}{1-\varepsilon} + \textstyle\sum_{s_j \in Li(N_i)} \Delta_{li}(s_j).$$

Solving for n' yields ($0 \leq \varepsilon < 1, T > 0$):

$$n' < \frac{1}{4T\varepsilon}\left(T(1-\varepsilon) - 2\textstyle\sum_{s_j \in Li(N_i)} \Delta_{li}(s_j)(1-\varepsilon^2)\right).$$

For $0 \leq \varepsilon < 1$, this constraint is satisfied if

$$n' < \frac{1}{4T\varepsilon}\left(T(1-\varepsilon) - 2\textstyle\sum_{s_j \in Li(N_i)} \Delta_{li}(s_j)\right).$$

Again, for $n' \leq n_{pf}$, this follows from Equation 3.3. This concludes the proof for case 2. □

Appendix C

Definitions and proofs for Chapter 4

As a property of $lin_{\bowtie}$, Lemma C.1 demonstrates that using the minimum-weight reflexive-transitive closure for the relations constituting $lin_{\bowtie}$ is not strictly necessary, and the usage of the reflexively-transitively reduced relations actually suffices from a formal standpoint.

Lemma C.1 (Sufficiency of reduced relations for defining $lin_{\bowtie}$). *Let $\bowtie^-$ be the map defined as, for all $\mathtt{x}, \mathtt{y} \in X$,*

$$\bowtie^-(\mathtt{x}, \mathtt{y}) = \min\big(\ \xrightarrow{(\cdot)}(\mathtt{x}, \mathtt{y})\ ,\ \Delta(\mathtt{x}, \mathtt{y})\ \big),$$

where $\xrightarrow{(\cdot)}$ is the reduced variable synchronization relation from Def. 2.10, and Δ is the reduced queuing capacity relation from Def. 4.1. It then holds that, for all $W \in (X \times V)^\omega$,

$$\big(\forall \mathtt{x},\mathtt{y} \in X\ .\ \#W'|_{\mathtt{x}} \leq \#W'|_{\mathtt{y}} + \bowtie^-(\mathtt{x},\mathtt{y})\big) \iff \big(\forall \mathtt{x},\mathtt{y} \in X\ .\ \#W'|_{\mathtt{x}} \leq \#W'|_{\mathtt{y}} + \bowtie(\mathtt{x},\mathtt{y})\big).$$

Proof. $\Leftarrow$ is clear. $\Rightarrow$ follows from the fact that minimum-weight reflexive-transitive closure (Def. 2.14) combines addition for transitive pairs, which is mirrored by conjunction of inequations in ($m, n \in \mathbb{N}$)

$$\#W'|_{\mathtt{x}} \leq \#W'|_{\mathtt{y}} + m \wedge \#W'|_{\mathtt{y}} \leq \#W'|_{\mathtt{z}} + n \implies \#W'|_{\mathtt{x}} \leq \#W'|_{\mathtt{z}} + n + m,$$

and selection of the minimum element for multiple weights, again mirrored by conjunction of inequations in ($m, n \in \mathbb{N}$)

$$\#W'|_{\mathtt{x}} \leq \#W'|_{\mathtt{y}} + m \wedge \#W'|_{\mathtt{x}} \leq \#W'|_{\mathtt{y}} + n \implies \#W'|_{\mathtt{x}} \leq \#W'|_{\mathtt{y}} + \min(m, n).$$

□

For the definition of $lin_{\bowtie}$, using the minimum-weight reflexive-transitive closure of synchronization relations, $\xrightarrow{(\cdot)}{}^{*}$ and Δ^* has an important methodical advantage over the usage of reduced relations $\xrightarrow{(\cdot)}$ and Δ: The rewriting of an SSDL program to its linearization-matching form, as described in Section 4.1.2, introduces a number of variable copies in order to match a given distribution to threads and 1-channels. However, the introduction such variable copies typically does not provide an improved understanding of the program, or its linearizations. One is typically interested in projections of the relations $\xrightarrow{(\cdot)}$ and Δ relating the original variables of an SSDL program, without their copies. For the reflexive-transitive variants $\xrightarrow{(\cdot)}{}^{*}$ and Δ^*, the projection onto the original variables is immediately clear by removing pairs referring to variable copies. For the reduced relations, on the other hand, removing pairs referring to copies would not yield the correct result.

Lemma 4.3 (Concatenation of linearized words is linearized word). *Let $w, w' \in (X{\rightarrow}V)^*$ be synchronous words. Then for all $W, W' \in (X \times V)^*$,*

$$W \in lin_{\bowtie}(w) \;\wedge\; W' \in lin_{\bowtie}(w') \implies (W \mathbin{.} W') \in lin_{\bowtie}(w \mathbin{.} w')$$

Proof. The first constraint of Def. 4.5, $((W \mathbin{.} W')|_{\mathtt{x}} = (w \mathbin{.} w')|_{\mathtt{x}})$, is clearly satisfied by $W \mathbin{.} W'$. For the second constraint, $\#(W \mathbin{.} W')|_{\mathtt{x}} \leq \#(W \mathbin{.} W')|_{\mathtt{y}} + \bowtie(\mathtt{x}, \mathtt{y})$, we note that for all $\mathtt{z} \in X$, $\#W|_{\mathtt{z}}$ is equal to $\#w$, so $\#(W \mathbin{.} W')|_{\mathtt{z}} = \#W'|_{\mathtt{z}} + \#w$, and hence the second constraint holds for $W \mathbin{.} W'$ iff it holds for W'. □

Lemma 4.4 (Linearized word with 1-synchronized variables is concatenation of linearized words). *Let X' be a 1-synchronized variable set. Let $w \in (X'{\rightarrow}V)^*, w' \in (X'{\rightarrow}V)^\omega$ be synchronous words. Then for all $W'' \in (X' \times V)^\omega$,*

$$W'' \in lin_{\bowtie}(w \mathbin{.} w') \implies \exists W \in lin_{\bowtie}(w), W' \in lin_{\bowtie}(w') \;.\; W \mathbin{.} W' = W''$$

Proof. Let W''' be the prefix of W'' such that $\#W''' = |X'| \cdot \#w$, where $|X'|$ is the size of X'. We can deduce that $\#W'''|_{\mathtt{x}} = \#w$ for all $\mathtt{x} \in X'$, as all other possibilities for W''' would violate the 1-synchronization constraint for X'. We can further deduce that $W'''|_{\mathtt{x}} = w|_{\mathtt{x}}$ for all $\mathtt{x} \in X'$, which is the first required constraint for $W \in lin_{\bowtie}(w)$ according to Def. 4.5. The second constraint holds for all prefixes of $W'' \in lin_{\bowtie}(w \mathbin{.} w')$ by definition, and hence for $W''' \sqsubseteq W''$. So there exists a $W = W'''$ such that $W \in lin_{\bowtie}(w)$, and $W' \in lin_{\bowtie}(w')$ follows symmetrically. □

Lemma 4.5 (Existential and universal acceptance of $lin_{\bowtie}(\sigma)$-languages for $Lin(A)$). *Let A be a synchronous automaton, and let $Lin(A)$ be its linearized counterpart. For a given $\sigma \in (X_A \to V)$, we write $\exists W_\sigma$ short for $\exists W_\sigma \in lin_{\bowtie}(\xrightarrow{(.)}{}^*_A, \sigma)$, and symmetrically for $\forall W_\sigma$. Then the following holds, for all $\sigma \in (X_A \to V)$, for all $s, s' \in S$,*

$$1.\quad \exists W_\sigma.(s,\emptyset) \xmapsto[Lin(A)]{W_\sigma}{}^* (s',\emptyset) \implies s \xmapsto[A]{\sigma} s'$$
$$2.\quad \forall W_\sigma.(s,\emptyset) \xmapsto[Lin(A)]{W_\sigma}{}^* (s',\emptyset) \impliedby s \xmapsto[A]{\sigma} s'$$

Proof. For statement 1., we show the equivalent statement

$$\neg\left(s \xmapsto[A]{\sigma} s'\right) \implies \neg\left(\exists W_\sigma.(s,\emptyset) \xmapsto[Lin(A)]{W_\sigma}{}^* (s',\emptyset)\right).$$

We write $\rightsquigarrow^*_A = Unweight(\xrightarrow{(.)}{}^*_A)$ for the local variable dependency order of A. If there is no transition $s \xmapsto[A]{\sigma} s'$, then by the construction procedure of $Lin(A)$ from A, any W' such that $(s,\emptyset) \xmapsto[Lin(A)]{W'}{}^* (s',\emptyset)$ will be a linearization of some symbol $\sigma' \neq \sigma$. But then clearly the constraint $\forall \mathtt{x} \in X_A.(W'|_{\mathtt{x}} = \sigma|_{\mathtt{x}})$ from the definition of $lin_{\bowtie}$ in Def. 4.5 cannot hold, thus $W' \notin lin_{\bowtie}(\xrightarrow{(.)}{}^*_A, \sigma)$.

Statement 2. can be similarly deduced by a comparison of the construction of $Lin(A)$ with $lin_{\bowtie}(\rightsquigarrow^*_A, \sigma)$: We first note that, by the construction rule for $Lin(A)$, on any transition path from $(s,\emptyset)$ to $(s',\emptyset)$ in $Lin(A)$, all possible sequences of all symbols $\gamma \in \sigma$, $\gamma \in X \times V$, in an order consistent with $\rightsquigarrow^*_A = Unweight(\xrightarrow{(.)}{}^*_A)$ are accepted by $Lin(A)$. On the other hand, according to Def. 4.5, $lin_{\bowtie}(\xrightarrow{(.)}{}^*_A, \sigma)$ allows precisely those words W with

$$\begin{array}{ll} \forall \mathtt{x} \in X_A. & (W|_{\mathtt{x}} = \sigma|_{\mathtt{x}}), \text{ and} \\ \forall \mathtt{x} \in X_A. \forall\, W' \in \sqsubseteq^{-1}(W). & \#W'|_{\mathtt{x}} \leq \#W'|_{\mathtt{y}} + \bowtie(\rightsquigarrow^*_A, \mathtt{x}, \mathtt{y}). \end{array}$$

So the *first constraint* mandates that the sequence of symbols $\gamma_1 \centerdot \gamma_2 \centerdot \ldots \centerdot \gamma_n = W$ corresponds precisely to the macrosymbol $\{\gamma_1, \gamma_2, \ldots, \gamma_n\} = \sigma$. For the *second constraint*, we note that for all $\mathtt{x}, \mathtt{y} \in X_A$, $\bowtie(\rightsquigarrow^*_A, \mathtt{x}, \mathtt{y} \leq 1$ by Def. 4.1, so $\mathtt{x} \rightsquigarrow^* \mathtt{y} \Leftrightarrow \mathtt{x} \xrightarrow{0}{}^* \mathtt{y}$, and $\mathtt{x} \not\rightsquigarrow^* \mathtt{y} \Leftrightarrow \mathtt{x} \xrightarrow{1}{}^* \mathtt{y}$. That is, if $\rightsquigarrow^*_A$ forces symbol $\mathtt{b}^{\mathtt{y}}$ to follow symbol $\mathtt{a}^{\mathtt{x}}$ in the construction of the transition path in $Lin(A)$ from $(s,\emptyset)$ to $(s',\emptyset)$ corresponding to $s \xmapsto[A]{\sigma} s'$, so does $\xrightarrow{(.)}{}^*{}_A$ in

the construction of $W \in lin_{\bowtie}(\rightsquigarrow_A^*, \sigma)$, and vice versa. So both constraints together ensure that all $W \in lin_{\bowtie}(\rightsquigarrow_A^*, \sigma)$ will be accepted by $Lin(A)$ on the transition path from $(s, \emptyset)$ to $(s', \emptyset)$. □

Lemma 4.6 (Equivalence of existential and universal acceptance of $lin_{\bowtie}(w)$-- languages for $Lin(A)$). *Let A be a synchronous automaton, and let $Lin(A)$ be its linearized counterpart. For a given $w \in (X_A \to V)$, we write $\exists W_w$ short for $\exists W_w \in lin_{\bowtie}(\xrightarrow{(.)}_A^*, w)$, and symmetrically for $\forall W_w$. Then the following holds, for all $\sigma \in (X_A \to V)$, for all $w \in (X_A \to V)^\omega$, for all $s, s' \in S$,*

$$1.\quad \exists W_\sigma.(s,\emptyset) \xmapsto[Lin(A)]{W_\sigma}{}^{*} (s',\emptyset) \iff \forall W_\sigma.(s,\emptyset) \xmapsto[Lin(A)]{W_\sigma}{}^{*} (s',\emptyset)$$

$$2.\quad \exists W_w.(s,\emptyset) \xmapsto[Lin(A)]{W_w}{}^{*} (s',\emptyset) \iff \forall W_w.(s,\emptyset) \xmapsto[Lin(A)]{W_w}{}^{*} (s',\emptyset)$$

We write $(s,\emptyset) \xmapsto[Lin(A)]{lin_{\bowtie}(\xrightarrow{(.)}_A^*, w)}{}^{*} (s',\emptyset)$ to denote both existential and universal acceptance of $lin_{\bowtie}(\xrightarrow{(.)}_A^*, w)$ by $Lin(A)$.

Proof. For statement 1., the $\Leftarrow$ direction is clear. The $\Rightarrow$ direction follows directly from combination of properties 1. and 2. of Lemma 4.5.

For statement 2., we first prove three auxiliary statements

$$2a.\qquad (s,\emptyset) \xmapsto[Lin(A)]{lin_{\bowtie}(\xrightarrow{(.)}_A^*, \varepsilon)}{}^{*} (s,\emptyset)$$

$$2b.\ \exists W_\sigma.\exists W_w\ (s,\emptyset) \xmapsto[Lin(A)]{W_\sigma}{}^{*} (s'',\emptyset) \wedge (s'',\emptyset) \xmapsto[Lin(A)]{W_w}{}^{*} (s',\emptyset) \Rightarrow \exists W_{\sigma w}.(s,\emptyset) \xmapsto[Lin(A)]{W_{\sigma w}}{}^{*} (s',\emptyset)$$

$$2c.\ \forall W_\sigma.\forall W_w\ (s,\emptyset) \xmapsto[Lin(A)]{W_\sigma}{}^{*} (s'',\emptyset) \wedge (s'',\emptyset) \xmapsto[Lin(A)]{W_w}{}^{*} (s',\emptyset) \Rightarrow \forall W_{\sigma w}.(s,\emptyset) \xmapsto[Lin(A)]{W_{\sigma w}}{}^{*} (s',\emptyset)$$

Statement 2a. is clear by the construction rule for Lin: if $s \xmapsto[A]{\varepsilon}{}^{*} s'$, then $s = s'$, hence $(s,\emptyset) \xmapsto[Lin(A)]{lin_{\bowtie}(\xrightarrow{(.)}_A^*, \varepsilon)}{}^{*} s'$ for $lin_{\bowtie}(\rightsquigarrow_A^*, \varepsilon) = \{\varepsilon\}$, and vice versa.

Statements 2b. and 2c. follow from the structure of $\xmapsto[Lin(A)]{(.)}{}^{*}$ and Lemma 4.4. The overall proof for 2. is obtained by induction over w, combining statements 1., 2a., 2b., and 2c.. □

Lemma 4.10 (Lin is compositional). *Let A_1, A_2 be synchronous automata. Then it holds that the linearized product of the linearized automata of A_1 and A_2 simulates the linearized automaton of the synchronous product of A_1 and A_2:*

$$Lin(A_1) \| Lin(A_2) \approx Lin(A_1 \| A_2).$$

Proof. We first note that the two alphabets coincide: the alphabet is equal to $((X_{A_1} \cup X_{A_2}) \times V)$ for both automata. To show the equality, we first define a map F from states of $Lin(A_1) \| Lin(A_2)$ to states of $Lin(A_1 \| \| A_2)$ as follows: for all $s_1 \in S_1$, $s_2 \in S_2$, $\sigma_1 \in (X_{A_1} \to V)$, $\sigma_2 \in (X_{A_2} \to V)$,

$$F((s_1, \sigma_1), (s_2, \sigma_2)) = ((s_1, s_2), \sigma_1 \cup \sigma_2)$$

Note that F is a bijection between the state sets: in the opposite direction, any state $((s_1, s_2), \sigma)$ of $Lin(A_1 \| \| A_2)$ can be mapped to the unique state $((s_1, \sigma|_{X_{A_1}}), (s_2, \sigma|_{X_{A_2}}))$ by the inverse of F, F^{-1}. We also observe that F maps the initial state of $Lin(A_1) \| Lin(A_2)$, $((s_{01}, \emptyset), (s_{02}, \emptyset))$, to the initial state of $Lin(A_1 \| \| A_2)$, $((s_{02}, s_{02}), \emptyset)$, and similar for accepting states S_a.

It remains to show that the two transition relations coincide up to bijection F. We shall establish the following property: for s_1 a state of A_1, s_2 a state of A_2, we write $s_{12\emptyset}$ for $((s_1, \emptyset), (s_2, \emptyset))$ and $s_{1'2'\emptyset}$ for $((s_1, \emptyset), (s_2, \emptyset))$. Then for all $W \in ((X_{A_1} \cup X_{A_2}) \times V)^*$ such that $\forall \mathtt{x} \in (X_{A_1} \cup X_{A_2}) \,.\, \#W|_{\mathtt{x}} = 1$,

$$s_{12\emptyset} \xmapsto[Lin(A_1) \| Lin(A_2)]{W}{}^{*} s_{1'2'\emptyset} \;\Leftrightarrow\; F(s_{12\emptyset}) \xmapsto[Lin(A_1 \| \| A_2)]{W}{}^{*} F(s_{1'2'\emptyset})$$

For automaton A_i, for $i \in \{1, 2\}$, we write $\rightsquigarrow_i^*$ for the variable dependency order $Unweight\big(\xrightarrow{(.)}{}^{*}_{A_i}\big)$. Similarly, we write $\rightsquigarrow_{12}^*$ for $Unweight\big(\xrightarrow{(.)}{}^{*}_{A_1 \| \| A_2}\big)$. Now note the following connection:

$Lin(A_1) \| Lin(A_2)$**:** We know from the definition of Lin that the set of $W_i \in X_i \times V)^*$ accepted on the transition from $(s_i, \emptyset)$ to $(s_i, \emptyset)$ in $Lin(A_i)$, $(s_i' \emptyset)$ is the set of $\rightsquigarrow_i^*$-linearizations of macrosymbol $\sigma_i \in (X_i \to V)$ such that $s_i \xmapsto[A_i]{\sigma_i} s_i'$. For constructing $Lin(A_1) \| Lin(A_2)$, the two individual automata A_1, A_2 are then composed using parallel composition $\|$.

$Lin(A_1 \| \| A_2)$**:** All $W_{12} \in ((X_{A_1} \cup X_{A_2}) \times V)^*$ accepted on the transition from $F(s_{12\emptyset})$ to $F(s_{1'2'\emptyset})$ in $Lin(A_1 \| \| A_2)$ are respective $\rightsquigarrow_{12}^*$-linearizations of macrosymbols $\sigma_{12} \in (X_i \to V)$ such that $(s_1, s_2) \xmapsto[A_1 \| \| A_2]{\sigma_{12}} (s_1', s_2')$. By the definition of the synchronous product $\| \|$, it clearly holds that $\rightsquigarrow_{12}^* = (\rightsquigarrow_1^* \cup \rightsquigarrow_2^*)^*$, the reflexive-transitive closure of the union of the individual orders.

So if we can show that composition $\|$ has the same effect as combining union and reflexive-transitive closure for variable dependency orders $\rightsquigarrow_1^*$, $\rightsquigarrow_2^*$, to construct $\rightsquigarrow_{12}^*$, we can effectively show equivalence of $s_{12\emptyset} \xmapsto[Lin(A_1) \| Lin(A_2)]{W}{}^{*} s_{1'2'\emptyset}$ and $F(s_{12\emptyset}) \xmapsto[Lin(A_1 \| \| A_2)]{W}{}^{*} F(s_{1'2'\emptyset})$.

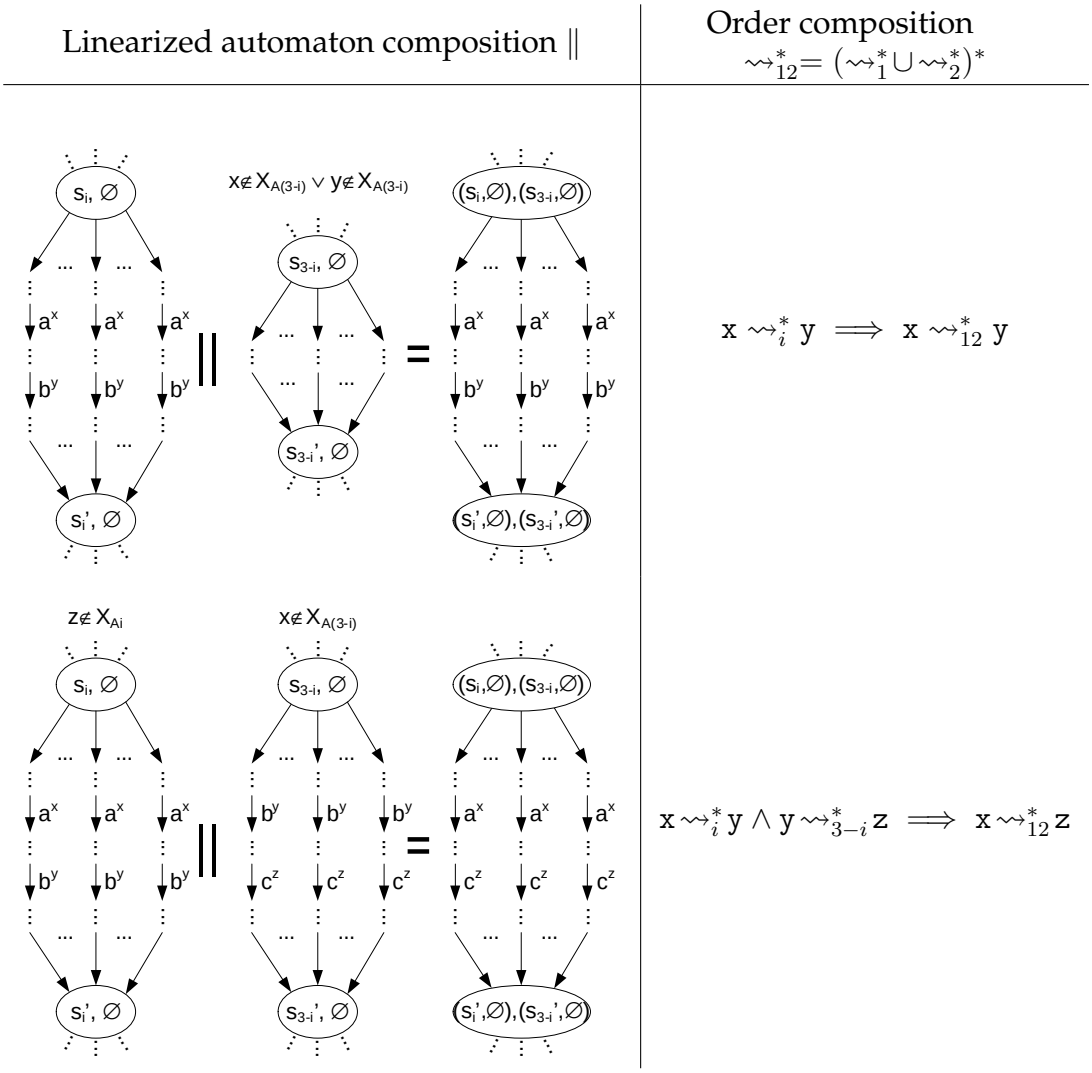


Figure C.1: Comparison of linearized automaton composition $\|$ and order composition $(\rightsquigarrow^*_1 \cup \rightsquigarrow^*_2)^*$

Now note the correspondance between the linearized automaton composition $\|$ and relation composition $\rightsquigarrow_{12}^* = (\rightsquigarrow_1^* \cup \rightsquigarrow_2^*)^*$, sketched in Fig. C.1 for a pair of linearized automata $Lin(A_i)$ and $Lin(A_{3-i})$, for all variables $\mathtt{x}, \mathtt{y}, \mathtt{z} \in X_{A_1} \cup X_{A_2}$, for all $\mathtt{a}, \mathtt{b}, \mathtt{c} \in V$.

$\mathtt{x} \rightsquigarrow_i^* \mathtt{y} \Rightarrow \mathtt{x} \rightsquigarrow_{12}^* \mathtt{y}$**:** In the upper row of Fig. C.1, for automaton $Lin(A_i)$, let $\mathtt{x}, \mathtt{y} \in X_{A_i}$ be a variable pair such that symbols $\mathtt{a}^{\mathtt{x}}$ always precede symbols $\mathtt{b}^{\mathtt{y}}$ on all transition paths from $(s_i, \emptyset)$ to $(s_i', \emptyset)$ of $Lin(A_i)$. From the structure of SSDL programs, and from the causality check described in Section 2.2.4, we can conclude that for the other linearized automaton $Lin(A_{3-i})$, it holds that either $\mathtt{x} \notin X_{A_{3-i}} \vee \mathtt{y} \notin X_{A_{3-i}}$, or that symbols $\mathtt{a}^{\mathtt{x}}$ and $\mathtt{b}^{\mathtt{y}}$ are not causally constrained on any transition path from $(s_{3-i}, \emptyset)$ to $(s_{3-i}', \emptyset)$. Then by definition of linearized composition $\|$, for the composed automaton $Lin(A_i) \| Lin(A_{3-i})$, symbols $\mathtt{a}^{\mathtt{x}}$ always precede symbols $\mathtt{b}^{\mathtt{y}}$ on all transition paths from $s_{12\emptyset} = ((s_i, \emptyset), (s_{3-i}, \emptyset))$ to $s_{1'2'\emptyset} = ((s_i', \emptyset), (s_{3-i}', \emptyset))$. This corresponds to the property $\mathtt{x} \rightsquigarrow_i^* \mathtt{y} \Rightarrow \mathtt{x} \rightsquigarrow_{12}^* \mathtt{y}$ for order composition.

$\mathtt{x} \rightsquigarrow_i^* \mathtt{y} \wedge \mathtt{y} \rightsquigarrow_{3-i}^* \mathtt{z} \Rightarrow \mathtt{x} \rightsquigarrow_{12}^* \mathtt{z}$**:** In the lower row of Fig. C.1, for automaton $Lin(A_i)$, let $\mathtt{x} \in X_{A_i}$, $\mathtt{y} \in X_{A_i} \cap X_{A_{3-i}}$ be a variable pair such that symbols $\mathtt{a}^{\mathtt{x}}$ always precede symbols $\mathtt{b}^{\mathtt{y}}$ on all transition paths from $(s_i, \emptyset)$ to $(s_i', \emptyset)$ of $Lin(A_i)$. For automaton $Lin(A_{3-i})$, symmetrically, let $\mathtt{z} \in X_{A_{3-i}}$ be a variable such that symbols $\mathtt{b}^{\mathtt{y}}$ always precede symbols $\mathtt{c}^{\mathtt{z}}$ on all transition paths from $(s_{3-i}, \emptyset)$ to $(s_{3-i}', \emptyset)$ of $Lin(A_{3-i})$. Then by definition of $\|$, for the composed automaton $Lin(A_i) \| Lin(A_{3-i})$, symbols $\mathtt{a}^{\mathtt{x}}$ always precede symbols $\mathtt{c}^{\mathtt{z}}$ on all transition paths from $s_{12\emptyset} = ((s_i, \emptyset), (s_{3-i}, \emptyset))$ to $s_{1'2'\emptyset} = ((s_i', \emptyset), (s_{3-i}', \emptyset))$. This corresponds to the property $\mathtt{x} \rightsquigarrow_i^* \mathtt{y} \wedge \mathtt{y} \rightsquigarrow_{3-i}^* \mathtt{z} \Rightarrow \mathtt{x} \rightsquigarrow_{12}^* \mathtt{z}$ for order composition.

It also holds that, for all $\mathtt{x}, \mathtt{z} \in X_{A_1} \cup X_{A_2}$,

$$\mathtt{x} \rightsquigarrow_{12}^* \mathtt{z} \;\Rightarrow\; \big(\exists i \in \{1,2\} \,.\, \mathtt{x} \rightsquigarrow_i^* \mathtt{z}\big) \wedge \big(\exists i \in \{1,2\} \,.\, \exists \mathtt{y} \in X_{A_1} \cap X_{A_2} \,.\, \mathtt{x} \rightsquigarrow_i^* \mathtt{y} \wedge \mathtt{y} \rightsquigarrow_{3-i}^* \mathtt{z}\big),$$

so our analysis captures all possible sources of $\mathtt{x} \rightsquigarrow_{12}^* \mathtt{z}$. Because automaton composition $Lin(A_1) \| Lin(A_2)$ and order composition $(\rightsquigarrow_1^* \cup \rightsquigarrow_2^*)^*$ thus yield the same transition paths from state $s_{12\emptyset}/F(s_{12\emptyset})$ to state $s_{1'2'\emptyset}/F(s_{1'2'\emptyset})$, it holds that, for all $W \in ((X_{A_1} \cup X_{A_2}) \times V)^*$ such that $\forall \mathtt{x} \in (X_{A_1} \cup X_{A_2}) \,.\, \#W|_{\mathtt{x}} = 1$,

$$s_{12\emptyset} \xmapsto[Lin(A_1)\|Lin(A_2)]{W}{}^{*} s_{1'2'\emptyset} \;\Leftrightarrow\; F(s_{12\emptyset}) \xmapsto[Lin(A_1|||A_2)]{W}{}^{*} F(s_{1'2'\emptyset}).$$

Consequently, $Lin(A_1) \| Lin(A_2)$ and $Lin(A_1 ||| A_2)$ have the same alphabet, states, transition relation, initial state, and accepting states up to bijection

F, so

$$Lin(A_1) \| Lin(A_2) \approx Lin(A_1 \| A_2).$$

□

The following is an auxiliary lemma that will be needed in the proof for Theorem 4.13.

Lemma C.2 (Indices for dependent symbol sets). *Let $w \in (X \to V)^\infty$ be a synchronous word. Let $X', X'' \subseteq X$ be two variable sets such that $X' \rightsquigarrow^* X''$. For some $i \in \mathbb{N}$, for some $W \in lin_{\bowtie}(w)$, let $I' \in \mathbb{N}$ be an index into W such that I' points to the W-symbol corresponding to some $\gamma' \in w_i|_{X'}$, and let $I''\mathbb{N}$ be an index into W such that I'' points the W-symbol corresponding to some $\gamma'' \in w_i|_{X''}$. Then for all valid choices of w, i, W, γ', γ'', it holds that $I' \leq I''$.*

Proof. We note that for $\mathtt{x}, \mathtt{y} \in X$, $\mathtt{x} \rightsquigarrow^* \mathtt{y} \Leftrightarrow \mathtt{x} \xrightarrow{0}{}^* \mathtt{y}$. Then the lemma follows from the constraint $\forall \mathtt{x}, \mathtt{y} \in X \,.\, \#W|_{\mathtt{x}} \leq \#W|_{\mathtt{y}} + \bowtie(\mathtt{x}, \mathtt{y})$ in the definition of $lin_{\bowtie}$ in Def. 4.5. □

Theorem 4.13 (Soundness of $PVar$). *Let $\varphi \in LTL_{(X \times V)}$ be an LTL formula. Then φ is preservable if $PVar(\varphi) \neq \top$:*

$$PVar(\varphi) \neq \top \implies Pres(\varphi).$$

Proof. As a preliminary note, we observe that, for all $\varphi \in LTL_{(X \times V)}$, $Pres(\varphi)$ or

$$w \models\!\!\models (\varphi) \impliedby lin_{\bowtie}(w) \models (\varphi)$$

can be rewritten to

$$w \not\models\!\!\models (\varphi) \implies lin_{\bowtie}(w) \not\models (\varphi),$$

which in turn reduces to, according to the respective LTL semantics definitions for $\neg\varphi$,

$$w \models\!\!\models (\neg\varphi) \implies lin_{\bowtie}(w) \models (\neg\varphi).$$

But $\neg\varphi$ is in $LTL_{(X \times V)}$ just like φ, so the $\Leftarrow$ direction for $Pres(\varphi)$ follows from the proof for the $\Rightarrow$ direction for $Pres(\varphi')$ for all $\varphi' \in LTL_{(X \times V)}$. So we shall concentrate on the $\Rightarrow$ direction for $Pres$ in the following.

We write $Sound(\varphi) \Leftrightarrow PVar(\varphi) \neq \top \Rightarrow Pres(\varphi)$ for the soundness condition of the theorem, and show $Sound(\varphi)$ by structural induction:

$Sound(tt)$**:** $Pres(tt)$ is clear, hence $Sound(tt)$.

$Sound(\mathtt{a}^{\mathtt{x}})$**:** For preservation of atomic propositions $\mathtt{a}^{\mathtt{x}}$, we know from the constraint in the definitition of $lin_{\bowtie}$ in Def. 4.5 that, for a given $w \in (X \rightarrow V)^{\infty}$, for all $W \in lin_{\bowtie}(w)$, for all $\mathtt{x} \in X$, $W|_{\mathtt{x}} = w|_{\mathtt{x}}$, and $W|_{\mathtt{x}}$ is nonempty. So clearly, for all $w \in (X \rightarrow V)^{\infty}$, $w \models\!\!\models \mathtt{a}^{\mathtt{x}} \implies lin_{\bowtie}(w) \models \mathtt{a}^{\mathtt{x}}$, so $Pres(\mathtt{a}^{\mathtt{x}})$ and $Sound(\mathtt{a}^{\mathtt{x}})$.

$Sound(\varphi) \implies Sound(\neg\varphi)$**:** $Pres(\varphi) \implies Pres(\neg\varphi)$ follows from the LTL semantics of $\neg$, and from $P \Rightarrow P' \implies \neg P \Rightarrow \neg P'$ for propositions P, P'.

$Sound(\varphi) \wedge Sound(\psi) \implies Sound(\varphi \vee \psi)$**:** $Pres(\varphi) \wedge Pres(\psi) \implies Pres(\varphi \vee \psi)$ follows from $(P \Rightarrow P') \wedge (Q \Rightarrow Q') \implies (P \vee P' \Rightarrow Q \vee Q')$ for propositions P, P', Q, Q'.

$Sound(\varphi) \wedge Sound(\psi) \implies Sound(\varphi \,\mathcal{U}\, \psi)$**:** It is either the case that

$$PVar(\varphi) = \top \vee PVar(\psi) = \top \vee (PVar(\psi) \not\rightsquigarrow^{*} PVar(\varphi)) \vee \varphi \notin \text{1-}CF \vee \neg Sync^{1}_{\bowtie}(PVar(\psi)),$$

then $Sound(\varphi \,\mathcal{U}\, \psi)$ trivially holds by Def. 4.8. Or it is the case that

$$PVar(\varphi) \neq \top \wedge PVar(\psi) \neq \top \wedge (PVar(\psi) \rightsquigarrow^{*} PVar(\varphi)) \wedge \varphi \in \text{1-}CF \wedge Sync^{1}_{\bowtie}(PVar(\psi)),$$

whereby we know by assumption that $Pres(\varphi)$ and $Pres(\psi)$. In order to satisfy $Sound(\varphi \,\mathcal{U}\, \psi)$, we have to show $Pres(\varphi \,\mathcal{U}\, \psi)$, written out as

$$w \models\!\!\models \varphi \mathcal{U} \psi \implies lin_{\bowtie}(w) \models \varphi \mathcal{U} \psi.$$

According to the synchronous LTL semantics, $w \models\!\!\models \varphi \,\mathcal{U}\, \psi$ corresponds to the statement

$$\exists i \in \mathbb{N} \,.\, w^{(i)} \models\!\!\models \psi \wedge w^{(1)} \models\!\!\models \varphi \wedge \cdots \wedge w^{(j)} \models\!\!\models \varphi \wedge \cdots \wedge w^{(i-1)} \models\!\!\models \varphi \text{ } (Pres\text{-LHS}).$$

where $1 \leq j < i$. This statement must be shown to imply, for all $W \in lin_{\bowtie}(w)$,

$$\exists I \in \mathbb{N} \,.\, W^{(I)} \models\!\!\models \psi \wedge W^{(1)} \models\!\!\models \varphi \wedge \cdots \wedge W^{(J)} \models\!\!\models \varphi \wedge \cdots \wedge W^{(I-1)} \models\!\!\models \varphi \text{ } (Pres\text{-RHS}).$$

By assumption $Pres(\psi)$ and by $w^{(i)} \models\!\!\models \psi$, it follows that $lin_{\bowtie}(w^{(i)}) \models\!\!\models \psi$, and $lin_{\bowtie}(w^{(i)})|_{PVar(\psi)} \models\!\!\models \psi$ by Lemma 4.12.

Now for some word $w \in (X \rightarrow V)^{\infty}$, let $W \in lin_{\bowtie}(w)$ be any one of its linearized words. According to Lemma 4.4, because $Sync^{1}_{\bowtie}(PVar(\psi))$ and $w|_{PVar(\psi)} = w' \,.\, (w^{(i)}|_{PVar(\psi)})$ for some $w' \in (PVar(\psi) \times V)^{*}$, there

exists an $I' \in \mathbb{N}$ such that $W^{(I')}|_{PVar(\psi)} \in lin_{\bowtie}(w^{(i)})|_{PVar(\psi)}$ and thus, $W^{(I')}|_{PVar(\psi)} \models\!\!\models \psi$. Lemma 4.12 yields that there must be also a corresponding $I \in \mathbb{N}$ such that $W^{(I)} \models\!\!\models \psi$. So we have shown that the first conjunct of $Pres$-RHS holds.

For the remaining conjuncts of $Pres$-RHS, let $\varphi^{1\text{-}CF}$ be a single-variable conjunctive term equivalent to φ. If $w^{(1)} \models\!\!\models \varphi^{1\text{-}CF} \wedge \cdots \wedge w^{(j)} \models\!\!\models \varphi^{1\text{-}CF} \wedge \cdots \wedge w^{(i-1)} \models\!\!\models \varphi^{1\text{-}CF}$, then for all variables $\mathtt{x} \in X$, the respective projection $w^{(j)}|_{\mathtt{x}}$ $\models\!\!\models$-satisfies those conjuncts in $\varphi^{1\text{-}CF}$ relating to $\mathtt{x}$ (Lemma 4.12). Because of the constraint $\forall \mathtt{x} \in X \,.\, W|_{\mathtt{x}} = w|_{\mathtt{x}}$ in the definition of $lin_{\bowtie}$ in Def. 4.5, and because of the coincidence of $\models\!\!\models$ and $\models$ for single-variable projections of words, for all $W' \in lin_{\bowtie}(w)$, for all variables $\mathtt{x} \in X$, for all $1 \leq j < i$, the respective projection $W'^{(j)}|_{\mathtt{x}}$ $\models$-satisfies those conjuncts in $\varphi^{1\text{-}CF}$ relating to $\mathtt{x}$. Consequently, because $\varphi^{1\text{-}CF}$ is by definition conjunctive, for $1 \leq J < I''$, $W'^{(J)}$ $\models$-satisfies $\varphi^{1\text{-}CF}$, where I'' is some upper bound. But $PVar(\psi) \rightsquigarrow^* PVar(\varphi)$, so by applying Lemma C.2 to the symbol w_i, it is clear that $I \leq I''$, where I is the index into W from the proof for the first conjunct of $Pres$-RHS. Consequently,

$$\exists I \in \mathbb{N} \,.\, W^{(I)} \models\!\!\models \psi \wedge W^{(1)} \models\!\!\models \varphi \wedge \cdots \wedge W^{(J)} \models\!\!\models \varphi \wedge \cdots \wedge W^{(I-1)} \models\!\!\models \varphi,$$

so we are done with the proof for $Pres(\varphi \,\mathcal{U}\, \psi)$ under the given assumptions, and $Sound(\varphi \,\mathcal{U}\, \psi)$ follows.

□

Appendix D

The Brock-Ackerman anomaly

The following example is taken from Russell [Rus89]. Consider a process, shown on the left hand side of Fig. D.1, with a single input channel and a single output channel. The behavior of the process is as follows: It makes a top-level choice between two behaviors. The first behavior is to output a 0, then wait for an input token, and then output a 0. We summarize this as $0; \texttt{read}; 0$. In the same notation, the other possible behavior is $\texttt{read}; 0; 1$. The input/output relation is derived as:

Input	Output
ε	0 or ε
nonempty	00 or 01

The second process on the right hand side of Fig. D.1 can make a three-way choice between the first two possible behaviors of the first processes, and the behavior $0; \texttt{read}; 1$. Clearly, the I/O relation is the same as for the first process.

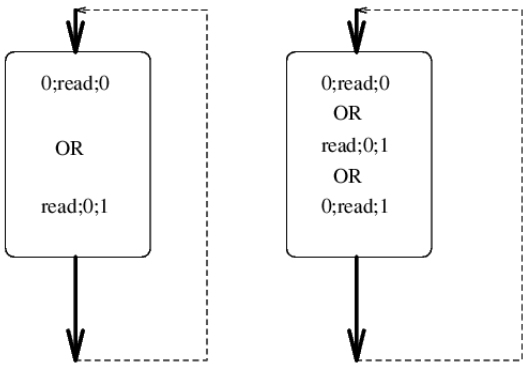


Figure D.1: Example for untimed dataflow network with nondeterminism

Now suppose that both processes feed back their outputs as inputs, shown by the dotted lines. Modified in such a way, both processes have only a single output, and no input. In the fed-back version, the first process can produce either ε or 00, while the second can produce ε, 00, or 01. Thus, two processes have the same input/output relation, but when placed in identical contexts, the respective composite processes have different input/output relations. In a sense, the input/output relation fails to capture the different causal dependencies in statements $\texttt{read}; 0; 1$ and $0; \texttt{read}; 1$.

Note that the Brock-Ackermann anomaly is a specific case of the "full abstraction" problem, which is usually stated as follows: A semantics is called *fully abstract* iff all equivalent terms, programs, or input-output behaviors are identified, i. e. mapped to the same semantic value. Equivalence is defined with respect to some relevant observation, such as convergence behavior in all possible term contexts for terms, or equivalent semantics in all possible contexts for data-flow processes [Win93]. A model is (informally) called *natural* if its structure is understandable without resorting to explicit notions of e. g. program equivalences or orderings. For instance, models built by using standard constructions of denotational semantics are generally considered natural. The "full abstraction" problem can then be summarized as synthesizing and finding natural models for given languages which are also fully abstract, i. e. formally adequate. For many programming languages, the standard techniques of denotational semantics yield natural models that are too concrete [Sto88]. The Brock-Ackermann anomaly, conversely, may be seen as an instance of a *too abstract* model: the relational semantics identifies behaviors which, operationally, can be distinguished by some context.

Bibliography

[AAR05] U. Assman, M. Aksit, and A. Rensink, editors. *Model Driven Architecture*, volume 3599 of *LNCS*. Springer Verlag, Berlin, 2005. European MDA Workshops: Foundations and Applicatios, MDAFA 2003 and MDAFA 2004 Twente, The Netherlands, June 2003 and Linkoping, Sweden, June 2004 Revised Paper selection.

[ABG95] T. P. Amagbegnon, L. Besnard, and P. Le Guernic. Implementation of the Data-flow Synchronous Language Signal. In *Proceedings of the ACM Symposium on Programming Languages Design and Implementation (PLDI'95)*, pages 163–173. ACM, 1995.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AGLS01] Rajeev Alur, Radu Grosu, Insup Lee, and Oleg Sokolsky. Compositional refinement for hierarchical hybrid systems. In *Proceedings 3rd International Workshop on Hybrid Systems: Computation and Control (HSCC '00)*, volume 2034 of *LNCS*. Springer Verlag, 2001.

[Alb79] A. J. Albrecht. Measuring application development productivity. In *Proceedings IBMGUIDE/SHARE Applications Development Symposium*, California, USA, 1979.

[Arm96] J. L. Armstrong. Erlang - A survey of the language and its industrial applications. In *Proceedings of the symposium on industrial applications of Prolog (INAP96)*, 1996.

[AUT03] AUTOSAR consortium. *AUTOSAR consortium homepage*, 2003. URL: `http://www.autosar.de`.

[AW77] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20(7):519–526, July 1977.

[BA81] J. Dean Brock and William B. Ackerman. Scenarios: A model of non-determinate computation. In *Proceedings of the International Colloquium on Formalization of Programming Concepts*, pages 252–259, London, UK, 1981. Springer-Verlag.

[BBR+05] A. Bauer, M. Broy, J. Romberg, Bernhard Schätz, P. Braun, U. Freund, N. Mata, R. Sandner, and D. Ziegenbein. AutoMoDe: Notations, Methods, and Tools for Model-Based Development of Automotive Software. In *Proceedings of the 2005 SAE World Congress*, Detroit, MI, 2005. Society of Automotive Engineers.

[BBRN05] I. Broster, A. Burns, and G. Rodríguez-Navas. Timing analysis of real-time communication under electromagnetic interference. *Real-Time Systems*, 30(1–2):55–81, May 2005.

[BCE+03] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 2003.

[BCG+02] A. Benveniste, P. Caspi, P. Le Guernic, H. Marchand, J.-P. Talpin, and S. Tripakis. A protocol for loosely time-triggered architectures. In *Proceedings of EMSOFT 2002*. Springer-Verlag, 2002.

[BCGH94] A. Benveniste, P. Caspi, P. Le Guernic, and N. Halbwachs. Data-flow synchronous languages. In *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium*, pages 1–45, London, UK, 1994. Springer-Verlag.

[Ber00] G. Berry. The Foundations of ESTEREL. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000.

[BFH+92] A. Bouajjani, J.-C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 18:247–269, 1992.

[BH95] Ed Baroth and Chris Hartsough. Visual programming in the real world. In *Visual object-oriented programming: concepts and*

environments, pages 21–42. Manning Publications Co., Greenwich, CT, USA, 1995.

[BJK+05] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer, 2005.

[BKP84] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computation*, pages 51–63, April 1984.

[BR04] Andreas Bauer and Jan Romberg. Model-based deployment: From a high-level view to low-level code. In *Proceedings of the 1st International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES)*, Hamilton, Canada, June 2004.

[Bro86] M. Broy. A Theory for Nondeterminism, Parallelism, Communication, and Concurrency. *Theoretical Computer Science*, 45(1):1–61, 1986.

[Bro87] F. P. Brooks. No silver bullet: essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[Bro93] M. Broy. Interaction refinement – The easy way. In *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993.

[BRS00] P. Braun, M. Rappl, and J. Schäuffele. Softwareentwicklung für Steuergerätenetzwerke – Eine Methodik für die frühe Phase. In *VDI Tagung Elektronik im KFZ*, number 1547 in VDI-Berichte, Baden-Baden, 2000.

[BS01] M. Broy and K. Stølen. *Specification and development of interactive systems: Focus on streams, interfaces, and refinement*. Springer-Verlag New York, Inc., 2001.

[Cas92] Paul Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94(1):125–140, 1992.

[CCGJ97] B. Caillaud, P. Caspi, A. Girault, and C. Jard. Distributing automata for asynchronous networks of processors. *European Journal of Automation*, 31(3):503–524, 1997.

[CCL91] M. Chen, Y. Choo, and J. Li. Crystal: Theory and Pragmatics of Generating Efficient Parallel Code. In *Parallel Functional Languages and Compilers*, Frontier Series, chapter 7, pages 255–308. ACM Press, 1991.

[CCM+03a] P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating discrete-time simulink to lustre. In *EMSOFT 2002*, volume 2855 of *LNCS*, pages 84–99, 2003.

[CCM+03b] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: A layered approach for distributed embedded applications. In *Proceedings of the 2003 ACM SIGPLAN conference on Languages, Compilers, and Tools for Embedded Systems*, pages 153–162. ACM Press, 2003.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CGP99] P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Transactions on Software Engineering*, 25(3):416–427, May/June 1999.

[Cla89] D. Clark. HIC: An operating system for hierarchies of servo loops. In *Proceedings of IEEE International Conference on Robotics and Automation*, 1989.

[CP96] P. Caspi and M. Pouzet. Synchronous Kahn Networks. In *International Conference on Functional Programming*, pages 226–238, 1996.

[dBOP+00] L. du Bousquet, F. Ouabdesselam, I. Parissis, J.-L. Richier, and N. Zuanon. Specification-based testing of synchronous software. In *Proceedings 5th Int'l Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, April 2000.

[DGG97] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.

[DR95] V. Diekert and G. Rozenberg, editors. *The Book of Traces*. World Scientific, Singapore, 1995.

[DS02] L.A.J. Dohmen and L.J. Somers. Experiences and Lessons Learned Using UML-RT to Develop Embedded Printer Software. In M. Oivo and S. Komi-Sirviö, editors, *Proceedings of PROFES 2002*, volume 2559 of *LNCS*, pages 475–484, 2002.

[dSP] dSPACE GmbH. *dSPACE Homepage*. URL: `http://www.dspace.com/`.

[Dud04] F. Dudenhöffer. Elektronik-Ausfälle: Tendenz steigend. *Hanser Automotive Electronics + Systems*, April 2004.

[E+97] J. Eisenmann et al. Entwurf und Implementierung von Fahrzeugsteuerungsfunktionen auf Basis der TITUS Client Server Architektur. In *Systemengineering in der Kfz-Entwicklung*, number 1374 in VDI Berichte, pages 309–325, 1997.

[Eis05] U. Eisemann. Guidelines for a Model-based Development Process with Automatic Code Generation. In *Proceedings of the 3rd Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER3)*, Paderborn, Germany, 2005. Heinz-Nixdorf Institut.

[Ele05] Focus: Qualität + Diagnose. *elektronik industrie*, July 2005. URL: `http://www.elektronik-industrie.de`.

[ETA] ETAS Engineering Tools GmbH. *ETAS Homepage*. URL: `http://en.etasgroup.com/`.

[Ets01] K. Etschberger, editor. *CAN Controller Area Network - Grundlagen, Protokolle, Bausteine, Anwendungen*. Hanser, 2001.

[Fle] FlexRay Group. *FlexRay Group Homepage*. URL: `http://www.flexray-group.org`.

[FMHH01] T. Führer, B. Müller, F. Hartwich, and R. Hugel. Time triggered CAN (TTCAN). In *SAE World Congress*, Detroit, 2001. SAE number 2001-01-0073.

[GEB03] J. Guo, S. H. Edwards, and D. Borojevich. Comparison of dataflow architecture and Real-Time Workshop Embedded Coder in power electronics system control software design. In *CPES 2003 Power Electronics Seminar and NSF/Industry Annual Review*, 2003.

[GP92] T. R. G. Green and M. Petre. When Visual Programs are Harder to Read than Textual Programs. In *Human-Computer Interaction: Tasks and Organisation, Proceedings ECCE-6*, 1992.

[GS88] J. B. Goodenough and L. Sha. The priority ceiling protocol: A method for minimizing the blocking of high priority ada tasks. In *IRTAW '88: Proceedings of the second international workshop on Real-time Ada issues*, pages 20–31, New York, NY, USA, 1988. ACM Press.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.

[HHK01] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *Proceedings of EMSOFT 2001*. Springer-Verlag, 2001.

[HMP92] T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *ICALP '92: Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, pages 545–558, London, UK, 1992. Springer-Verlag.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, 1985.

[HPS02] H. Huang, P. Pillai, and K. Shin. Improving wait-free algorithms for interprocess communication in embedded realtime systems, June 2002.

[HRR91] N. Halbwachs, P. Raymond, and C. Ratel. Generating Efficient Code from Data-Flow Programs. In *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming*, Passau, Germany, 1991. Springer Verlag.

[HSE97] F. Huber, B. Schätz, and G. Einert. Consistent Graphical Specification of Distributed Systems. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *Industrial Applications and Strengthened Foundations of Formal Methods (FME'97)*, LNCS 1313, pages 122–141. Springer Verlag, 1997.

[IBM03] IBM Rational. *Rational Rose RealTime, C++ Reference*, 2003.06.00 edition, 2003.

[Int96] International Telecommunications Union (ITU), Geneva, Switzerland. *ITU-T Recommendation Z.120: Message Sequence Chart (MSC)*, October 1996.

[JD75] J.B.Dennis and D.P.Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the Second Annual Symposium on Computer Architecture*, pages 126–132. ACM, 1975.

[JHM04] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 2004.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of IFIP 74 Congress*, Amsterdam, 1974. North Holland.

[Kfz04] 17% der Autofahrer haben mit Elektronikproblemen ihres Fahrzeugs zu kämpfen. *Kfz-Elektronik*, May 2004. URL: `www.kfz-elektronik.de/article/b_040122.htm`.

[Kop92] Hermann Kopetz. Sparse Time versus Dense Time in Distributed Real-Time Systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems (ICDCS)*, pages 460–467, 1992.

[Kop97] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer, Boston, 1997.

[KP02] Raimund Kirner and Peter Puschner. International workshop on WCET analysis - summary. Research Report 12/2002, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2002.

[Lee03] Edward A. Lee. Model-Driven Development: From Object-Oriented Design to Actor-Oriented Design. In *Workshop on Software Engineering for Embedded Systems From Requirements to Implementation*, Chicago, IL, 2003. Invited talk.

[Lee05] E. Lee. Absolutely Positively On Time: What Would It Take? *IEEE Computer*, July 2005.

[Leu02] M. Leucker. *Logics for Mazurkiewicz Traces*. PhD thesis, RWTH Aachen, 2002. Aachener Informatik Berichte AIB-2002-10.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.

[LM87] E. Lee and D. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75:1235–1245, 1987.

[LT87] N. A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, 1987.

[Mac94] S. G. MacDonell. Comparative review of functional complexity assessment methods for effort estimation, 1994.

[Mil80] R. Milner. *A calculus of communicating systems*. Springer-Verlag, 1980.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992.

[MS83] J. McGraw and S. Skedzielewski. *SISAL—Streams and Iteration in a Single Assignment Language: Reference Manual*. Livermore National Laboratory, Livermore, CA, 1983.

[MT00] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[MW] The MathWorks Inc. *The MathWorks Inc. Homepage*. URL: `http://www.mathworks.com`.

[Nik90] R.S. Nikhil. Id reference manual. Technical Report CSG Memo 284-1, Massachusetts Institute of Technology, Laboratory for Computer Science, 1990.

[OMG05] Object Management Group. *Unified Modeling Language (UML) Superstructure*, August 2005. Version 2.0 formal/05-07-04.

[OSE01] OSEK VDX consortium. *OSEK/VDX Operating System Version 2.2*, 2001.

[PA92] D.E. Perry and A.L.Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[Par95] T. M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California at Berkeley, 1995.

[PBC05] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. In *Proceedings of the Fifth International Conference on Application of Concurrency to System Design, ACSD 2005*, 2005.

[PBM+93] J.P. Paris, G. Berry, F. Mignard, P. Couronne, P. Caspi, N. Halbwachs, Y. Sorel, A. Benveniste, T. Gautier, P. Le Guernic, F. Dupont, and C. Le Maire. Projet SYNCHRONE : les formats communs des langages synchrones. Technical Report RT-0157, INRIA, 1993.

[PMSB96] S. Poledna, Th. Mocken, J. Schiemann, and Th. Beck. ERCOS: An operating system for automotive applications. Research Report 21/1996, Technische Universität Wien, Institut für Technische Informatik, 1996.

[PT91] G. M. Papadopoulos and K. R. Traub. Multithreading: A revisionist view of dataflow architectures. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, volume 19, pages 342–351, New York, NY, 1991. ACM Press.

[RB04] J. Romberg and A. Bauer. Loose synchronization of event-triggered networks for distribution of synchronous programs. In *Proceedings 4th ACM International Conference on Embedded Software*, Pisa, Italy, September 2004.

[Ree95] H. J. Reekie. *Realtime Signal Processing*. PhD thesis, University of Technology at Sydney, Australia, 1995.

[RH05] G. Rosu and K. Havelund, editors. *Proceedings of the Fourth Workshop on Runtime Verification (RV'04)*, volume 113 of *Electronic Notes in Theoretical Computer Science*. Barcelona, Spain, 2005.

[Rus89] James R. Russell. Full abstraction for nondeterministic dataflow networks. In *Annual Symp. Foundations of Computer Science*, pages 170–177, 1989.

[RVA+98] J. Rufino, P. Veríssimo, G. Arroz, C. Almeida, and L. Rodrigues. Fault-tolerant broadcasts in CAN. In *28th International Symposium on Fault-Tolerant Computing Systems*, pages 150–159, Munich, Germany, June 1998. IEEE.

[RWNH98] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic Testing of Reactive Systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, 1998.

[SC85] A. P. Sistla and E. M. Clarke. Complexity of propositional temporal logics. *Journal of the ACM*, 32:733–749, 1985.

[SC04] N. Scaife and P. Caspi. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *Proceedings of ECRTS*, pages 119–126, 2004.

[Sel03] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

[SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[Soc04] Society of Automotive Engineers (SAE). *Architecture Analysis & Design Language (AADL)*, November 2004. SAE Standard No. AS5506.

[SPFR98] M. Saksena, A. Ptak, P. Freedman, and P. Rodziewicz. Schedulability analysis for automated implementations of real-time object-oriented models. In *IEEE Real-Time Systems Symposium*, 1998.

[SRL90] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.

[SRS+03] B. Schätz, J. Romberg, M. Strecker, O. Slotosch, and K. Spies. Modeling Embedded Software: State of the Art and Beyond. In *Proceedings of ICSSEA 2003 16th International Conference on Software and Systems Engineering and their Applications*, 2003.

[SS72] D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In *Proceedings Symposium on Computers and Automata*, pages 19–46. Wiley, 1972.

[Sta01] T. Stauner. *Systematic development of hybrid systems*. PhD thesis, Technische Universität München, 2001.

[Sto88] Allen Stoughton. *Fully abstract models of programming languages*. Pitman Publishing, Inc., Marshfield, MA, USA, 1988.

[Sys05] SysML Partners. *Systems Modeling Language (SysML) Specification version 1.0 alpha*, 2005. URL: `www.sysml.org`.

[Tan96] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall, 1996.

[TB94] K. Tindell and A. Burns. Guaranteeing message latencies on controller area network (CAN). In *Proceedings 1st International CAN Conference*, September 1994.

[THW01] W. Taha, P. Hudak, and Z. Wan. Directions in Functional Programming for Real(-Time) Applications. In *Proceedings of 2001 Workshop on Embedded Software, EMSOFT'01*, volume 2211 of *LNCS*, pages 185–203, 2001.

[TSSC05] S. Tripakis, C. Sofronis, N. Scaife, and P. Caspi. Semantics-preserving and memory-efficient implementation of inter-task communication under static-priority or edf schedulers. In *ACM Intl. Conference on Embedded Software (EMSOFT)*, 2005.

[Var01] M.Y. Vardi. Branching vs. linear time: Final showdown. In *ETAPS: European Joint Conference of Theory and Practice of Software*, 2001.

[Ves00] S. Vestal. Formal verification of the metah executive using linear hybrid automata. In *Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS)*, page 134, Washington, DC, USA, 2000. IEEE Computer Society.

[Wig01] U. Wiger. Four-fold increase in productivity and quality. In *FemSYS 2001*, Munich, Germany, 2001.

[Win93] Glynn Winskel. *The formal semantics of programming languages: An introduction*. MIT Press, Cambridge, MA, USA, 1993.

[WL88] J.L. Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, 1988.

List of Figures