The Automotive CASE

P. Braun, M. Broy, M.V. Cengarle, J. Philipps,
W. Prenninger, A. Pretschner, M. Rappl, R. Sandner

Institut für Informatik, TU München
Boltzmannstr. 3
85748 Garching, Germany

Abstract

The increasing functionality and complexity of modern automotive embedded systems demand new ways of mastering their intricacies. The standard response to this challenge is the use of abstractions, of appropriate structuring concepts, and, related, but less important, of suitable description techniques. Such ideas materialize in CASE tools. We shed light on CASE in the context of complex embedded systems, taking into account the different activities and phases of a development process. Clearly, integrated tool support is desirable for issues as different as requirements engineering, deployment, code generation, and validation. However, this is far from being simple since different levels of abstraction need to be interrelated. For instance, the embedding of generated code into its context of legacy systems, hardware, and operating systems requires dealing with both abstract models and an actual environment. We give our assessment of current and future CASE tools.

1 Introduction

With as many as 80 micro controllers in modern vehicles, engineers see themselves confronted with an ever increasing level of complexity. This comprises both the product and the development process. In addition to the definition of suitable, domain-specific processes, a lack of serious alternatives renders machine support for product development desirable. We will refer to this kind of machine support as CASE tools. Among other features, CASE tool functionality includes integrated support for all kind of specification activities including reuse, verification and validation, code generation and tracing as well as configuration and version management. Machine support for some of these aspects seems to be independent of the application domain,

while for others—e.g., modeling—, there is obviously a need for domain specific entities.

Our understanding of CASE tools exceeds classical IDEs. It is impossible to imagine the development of modern vehicles without machine support—just consider CAD for shaping parts of the vehicle, FEM simulation for flow analyses, and simulations of the driving behavior of entire cars. However, such tools are not the subject of this paper.

One desirable property of CASE tools is the integration of the different phases of a software development process, the integration of different levels of abstraction within each of the phases, and tracking facilities for the various stages of development of the system under development. The silver bullet, the all-inclusive CASE tool that allows the convenient handling of all difficulties, seems unrealistic. Integration of several CASE tools is therefore mandatory.

For a subset of the above desirable properties, the market offers solutions. Modeling mixed discrete-hybrid systems with tools like Matlab or ASCET-SD (ETAS) with associated (production) code generators like Beacon (ADI) or TargetLink (dSpace) is, for single control units and heavy restrictions on the modeling languages, successfully done in practice. However, there is a plethora of unsolved problems—such as the integration of different levels of abstraction, support for different communication buses in distrubuted systems, support for hardware platforms, etc.—that we sketch together with potential solutions in this paper. One of the key problems seems to be the following: demanding high levels of abstraction on the one hand and demanding support for code generation on the other hand, which includes aspects like communication with operating systems and hardware, seems to be contradictory.

CASE for automotive systems. The use of electronic control units in automobiles has a tradition of about thirty years now. The first electronic control unit was introduced to control fuel injection of a motor. This control unit came up with great expectations which were set with regard to the enhancement of the engine's performance, while reducing the fuel consumption at the same time. About ten years later, a programmable microprocessor based control unit was deployed for the first time: the Anti Blocking System (ABS). At that time nobody could expect that microprocessor based control units would be used in all kinds of application domains, including powertrain, chassis electronic, body car electronic, infotainment and driver assistance nowadays. In upper class cars, a vast number of control functions are executed on a network of about 80 control units, each fulfilling a dedicated task—a situation that is slightly different in avionic control systems where redundant full fledged micro processors like 68030 take care of different tasks at the same time.

Though innovative functions are key potentials for competitive advantage, their merit will be limited if they are not subject of a permanent cycle of innovation and change. Recent studies [24] show that the interest of the electronic content in the net product of a car will double till 2005. Further, to meet customers' wishes and requirements formerly isolated functions are growing together to car wide applications executed

on a distributed network of control units. Of course, despite this integration of functions and the growing complexity of the systems the car wide applications have to be easily maintainable, changeable and customizable according to the drivers' demands.

Along with the evolution of automotive embedded systems, the development processes of car vendors had to be refined. With the growing complexity of the realized systems, some well used practices exhibited some severe drawbacks. In the beginning, functions were developed in isolated development teams within one company. This worked well as long as the functions fulfilled their task in isolation. With the increasing number of functions and the growing amount of interactions, the integration of these functions failed. Problems like side effects and timing latencies arised. To overcome the difficulties of a distributed development of a system, the different development artifacts had to be communicated across the different development teams in order to gain a understanding of the mutual dependencies between subsystems. But most of the developed systems were not properly documented. In most cases, only the source code of realized functions could be exchanged, which was not sufficient to gain a deeper systems understanding. Abstract models which could help to constitute an improvement were used extremely rarely. The problems of a distributed development were reinforced when the purchase of components from third party suppliers started. In addition to the difficulties of integrating these components into existing systems, car vendors complained about a progressing lack of knowledge of their own systems.

About 15 years ago model based techniques were employed with great expectations to overcome recognized difficulties of current development processes. Especially in other engineering disciplines like electrical and mechanical engineering, model based techniques, such as CAD, FEM and hardware design tools were employed with a great success. Therefore various tools supporting visual modeling languages, configuration management tools, requirements management tools, and test tools were introduced. But the great heterogeneity of model based techniques, their abstractness and the strict focus of these tools on usually just one aspect of the development process limited the success and the effort of these tools. As a consequence car vendors started expensive integration projects to enhance the model based support of the overall development process. But all of these projects ended with an uncertain result. Step by step car manufacturers got a new awareness of their systems' architectures [10]. Partitioning the system's architecture in different abstraction levels, the introduction of a domain specific ontology, concepts for the formation of variants, and the understanding of model based configurations today seems a first step towards a successful deployment of model based techniques.

Notions such as signals, functions, electronic control units, real time operating systems, communication infrastructure, and processors now build up the automotive specific ontology that characterizes automotive embedded systems. Each of these notions constitutes a fragment of the architectural model at a certain level of abstraction. Signals, for instance, are elementary entities which can be exchanged between actors, sensors and control units. Each signal can be measured or computed from a

physical context. For the construction of architectural models, model based management of all signals occurring in a car is essential since their number goes far beyond ten thousand. Another important aspect of architectural models is the specification of data dependencies between control functions. Since functions are potentially distributable units that can be deployed on different control units, the consistent and the complete capture of model information in terms of data dependencies between functions supports the collaboration of distributed development teams. This happens by stating the required and offered signal interfaces which are the basis for the description of potential interaction patterns. Last but not least, the model based deployment of functions to different control units, and the model based connection of functions and control units to the technical infrastructure (processors, real time operating systems, communication infrastructure) are essential for an automotive specific support of the development process.

The attentive reader might argue that the listed notions of the automotive specific ontology are not unique in the automotive domain and that they are transferable to further application domains, e.g. avionics. And he might be right with this observation. Case Studies we have carried out in both application domains underpin this tight correlation. But each application domain owns characteristic factors which make them well distinguishable. In the automotive domain non-functional constraints exist, which limit the design space spanned by all possible architectural models. These constraints stem from political, economical, quality and technical requirements for automotive systems and imply automotive specific architectural models.

Technical constraints, for instance, may restrict the deployment of functions. Deployment of functions depends to a large extent on the availability of signals. Because most of the sensors and actuators do not have any logic for a linkage to a bus system, they are directly connected to the control unit. The deployment is therefore restricted w.r.t. the availability of signals. Their transmission is generally impeded by geometric restrictions due to the installation of the harness.

Quality constraints, as another example, can be discussed in relationship with upcoming X-by-wire systems. These safety relevant systems replace mechanically controlled processes by electrically controlled processes. On the one hand, these technologies allow the realization of comfortable driver assistance systems. On the other, hand these systems have to satisfy strict requirements on availability since there are no mechanical backup solutions. The requirements on availability affect the formation of architectural models with regard to a redundant realization of these safety critical systems.

Economical constraints are imposed according to the great amount of cars which are manufactured. Therefore, the cost of the used hardware has to be reduced. This leads to the installation of old-fashioned control units with low memory and processor resources. Sometimes these control units are not capable of executing all of the deployed functions. This again has influences on the specification of an architectural model.

Last but not least, political constraints such as road traffic acts influence the use of

technologies or require the implementation of certain systems. At this point, the controller for the regulation of the lightening distance can serve as an example.

Organization. The remainder of the paper, a critique of current CASE tools and a discussion of their future, is organized as follows. In section 2, we sketch the problems CASE tool developers have to face. This includes technical as well as economical and political aspects, and sometimes, these lead to conflicting conclusions. We outline an assessment of which problems appear solvable in the near future and which difficulties are likely to pose major obstacles. Section 3 is the core of the paper. It contains a survey of the state of the art, taking into account the challenges that have been identified in the previous Section. Section 4 concludes. Related work is cited in the context.

2 CASE Tool Challenges

CASE tool vendors—not only in the domain of automotive embedded systems—have a tendency to praise their tools as a response to the demands of increasing complexity in software and systems design. Conversely, there are a number of challenges that CASE tools must meet. In this section, we discuss two major streams of challenges, technical and political or economic ones.

2.1 Technical Challenges

Abstractions. Advances in programming languages are bound to the use of suitable abstractions [25]. Among others, this is illustrated by the invention of structured programming constructs, structured data types and type systems, modules, exceptions, monitors—and their equivalents—, automatic garbage collectors, communication infrastructure like CORBA[1], and, more recently, model-driven approaches to architecture like the OMG's MDA [30] initiative.

One of the most common misconceptions is that CASE tools for embedded systems should be *general* and equally well-suited for the development of internet appliances, signal processing in cellular telephones, switching systems for telecommunication and monolithic or distributed controllers in avionics or automotive electronics. Consequently, CASE tools typically have graphical description techniques that tend to focus on domain-unspecific description techniques for the different aspects of a system, like data, behavior, structure, or communication. It is, however, by no means certain that the abstractions behind these description techniques are better-suited for embedded systems than those of programming languages. There is, for instance, the danger that the various flavors of state machines found in CASE tools are used as an ill-structured graphical notation for the control flow in common programming languages ("spaghetti code"). In that case, modern IDEs for C, C++ or Ada are likely to offer higher productivity.

As long as commonly accepted abstractions—understood in terms of re-usable ontological entities—are not found, we deem domain specificity desirable. In fact, collaborating in a domain specific manner might well be the only way to identify generally applicable abstractions. It is unclear whether, for instance, the aspect of communication can be concisely described by the same description techniques both for CAN buses and time triggered architectures [30]. Common abstractions for the task scheduling of real-time systems are somewhat in conflict with abstraction for modular system structures [16]. For other aspects of modern automotive systems, such as fault tolerance, monitoring, auditing and maintenance, suitable abstractions are under-developed and little understood; it is no surprise that suitable abstractions for these aspects still remain to be found.

Code generation. A popular argument against the use of CASE tools is code quality, both in terms of memory requirements and execution speed; for automotive electronics, this argument is often accompanied by a reference to cost pressures that forbid the move to controllers with higher processing power. Although the code quality of modern tools has improved to a point that generated code has actually been deployed, code generators will need further improvement (unfortunately, many optimizations work only for a given compiler and target platform, which somewhat contradicts the use of high-level languages as a portable assembly language).

The problem of code generation from models in non-trivial. Models in CASE tools are used because they are abstractions, because they hide details. On the other hand, in order to become operational, code generation from models requires a degree of precision that is akin to that of code. This approach is, for instance, implemented in ASCET-SD (ETAS) for single ECUs or within Matlab with associated code generators like Beacon or TargetLink. Models must thus integrate different levels of abstractions, they must deal with both abstract descriptions as well as concrete interfaces with operating systems or bus infrastructures. That this is at least feasible for some aspects is, for instance, illustrated by ASCET-SD or the language Giotto [13]. Suitable descriptions of platforms remain to be found; tailoring of CASE tools to specific needs remains to be investigated.

Tool integration. Third, unless one is convinced of the possibility of one tool that solves all problems, the integration of different tools becomes an issue. Building explicit meta models and integrating different meta models is a difficult challenge. This is particularly if the issue of semantics is taken into account. The degree of formality of a semantics does not seem to be point, rather, we consider its simplicity a necessary prerequisite.

Note that a purely syntactic standardization as in the UML does not suffice.

Certification. The development of X-by-wire technologies for automotives puts high demand on the quality of code and development process. The field of avionics makes

use of elaborate certification processes, like DO-178B; it is likely that similar (albeit streamlined) approaches will be established for automotive electronics.

The avionics standards in general require the entire tool chain to be certified, which is a difficult problem. The short-term solution to this problem seems to be a focus on support for the certification of generated code rather than certification of the tools. Implications for the process are unclear. Certification of code also puts demands on the readability and traceability of the generated code, which forbids many optimization techniques.

In any case, it certainly is possible to integrate model-based code with code certification processes. Preliminary work for avionics has been done in the project MOBASIS [9]; it can easily be generalized for automotive electronics.

Traceability. Fifth, tracing the software development is yet unsolved by CASE tools. While this is also true for traditional code-centered development, the issue obviously becomes more prominent for systems of increasing complexity. This also includes requirements tracing. Support by means of tools like DOORS only is a first, if popular, step. It remains insufficient, unless one is able to structure requirements in a given domain in a consistent and reproducible manner, and unless the very nature of requirements in automotive systems is understood.

2.2 Political and Economic Challenges

In addition to the more technical issues mentioned above, CASE tool vendors are faced with a second kind of difficulties.

Vendor lifetime. The past years have seen the rise and fall of many tool vendors: Typically, companies that use CASE tools have a much higher lifetime than companies which produce them. The fear of vendor lock-in is a major reason that CASE tools are not used more widely. Small companies with uncertain prospects are not likely to be appointed as partners for large projects. Note that a similar situation for compiler vendors had been circumvented by standardization of programming language syntax and semantics; such a process, however, is unlikely to occur for embedded system description techniques in the near future.

Market size. In spite of the number of CASE tools currently available for embedded systems design, both market penetration of each particular tool and tool acceptance by embedded systems engineers is still rather low. From the point of view of the tool vendors, the CASE tool market—particularly for automotive systems—, certainly is not a mass market. The number of licenses is naturally restricted. This does not improve when domain-specific tools and description techniques are advocated. Building CASE tools, however, is a costly undertaking. There are two possibilities:

one can demand high prices for CASE tools which, in view of their currently limited capabilities, does not increase their attractivity. Alternatively, tool vendors can try to litigate expensive consulting and maintenance contracts. Outsourcing the mastery of process-critical methods and tools clearly is a critical option.

Supplier management. Typically, automotive controllers do not come from a single supplier; different controllers come from different suppliers, sometimes even the supplier changes for different controller series. The distribution of controller requirements and the collection of the final controller software is difficult unless tools and their description techniques are heavily standardized. Similar to the problem of tool integration mentioned above, a purely syntactic standardization as in the UML does not suffice.

Developer education. Most of the benefits of development with CASE tools come from the higher abstraction level of their description languages; consequently, the pure size of the descriptions is reduced, and—in theory—so is the size of the development team. However, in practice abstraction works only up to a point; then it is difficult to find developers that have the necessary education to skillfully work with the abstractions. This effect can be observed with the highly abstract functional programming languages Haskell [32] and ML [20]; although they are taught as a first programming language in many universities world-wide, their actual use is limited mainly to universities and research institutions —a phenomenon, which in view of the development of the Java language, cannot simply be blamed on the assumed "inefficiency" of these languages.

For CASE tools, this means that increasing the degree of abstraction requires engineers to be able to see a system at a high level, while, at least currently, they must not forget the implications at the level of the actually deployed system. Of course, this problem is not inherently bound to the use of tools but also applies to code-centered processes. However, structuring complex systems is a demanding activity, and experts are scarce. Less excellently educated engineers are unlikely to embrace CASE if they do not see the need nor understand the underlying design principles.

3 State of the art of CASE tools

Case tools provide a valuable support for the presentation and analysis of models. Their characteristics as listed above have been tackled to a certain extent by different existing CASE tools. In this section we schematically present these tools by addressing which concepts have been developed so far as well as the current solutions in the market, and discuss their strengths and weaknesses.

3.1 Developed concepts

The tasks and characteristics of (potential) CASE tools can be gathered in concepts as detailed below.

Graphical modeling techniques

Today's CASE tools, in particular those applied in the automotive domain, are dominated by graphical description techniques, and this not just only because of the impact of object orientation. The goal of using diagrams is to have at one's disposal a language-neutral modeling instrument without being tied to the intricacies of a programming language. This makes it easier to present a system under development to (potential) customers and discuss with them their visions, needs, and critics. Most used techniques are class or block diagrams for structure modeling, state based techniques such as Petri nets or statecharts for behavior and control flow modeling, and message sequence charts or sequence diagrams for interaction modeling.

Views and model management

A major approach of CASE tools to reduce the complexity of system development is the decomposition of a system description in views described by the modeling techniques discussed above. Views concentrate on aspects relevant for particular development taskts or provide abstractions suitable for specific development phases or tasks.

Almost all CASE tools provide at least views for modeling the logical structure and behavior of a system. Further views frequently provided are diagrams for the specification of component interaction, use cases, deployment, and related implementation specific structures.

Because of dependencies between development steps, it is natural that these views are not orthogonal: For example, a behavior model belongs to a component in the structure view, or an interaction refers to a component also specified in the behavior view. This redundancy requires mechanisms for ensuring consistency, either by consistency checks or by an appropriate propagation of changes. Such mechanisms are either based on explicit consistency conditions between the internal representation of its model elements in the tool or by a mapping of the views into an abstract meta model which avoids redundant representation.

Most CASE tools provide model management functionalities for versioning, further configuration management, and group work support such as change logs and access control.

Model analysis

Once a software system, be this prototypical or not, has been developed, a deep comprehension of the generated system and also tuning of the model might be advisable. By comprehension we mean an understanding of the system such that the designer knows to what extent the system is correct and fulfills its requirements.

Given an object-oriented model expressed for instance with the help of UML, there are tools that can generate package dependency diagrams, recover and display inheritance trees, or track any other kind of relation as a method parameter or a return value, such as dependencies, associations, realizations, and class usage. Tools exist that claim to have the ability to use integrated simulator for performance analysis of real-time systems or to provide export facilities (e.g. based on XML) in order to use an external simulator. To export structure of a model is relatively easy, a major challenge is to export behavior as well.

Quality assurance features, like for instance audits that help detect when the design is poor and metrics, spare developers from searching code for errors. Built-in unit testing, moreover, helps uncover problems during the coding process.

Automation of development

Ideally, once a model is constructed, a program is generated automatically. As a matter of fact, parts of the model can indeed be generated automatically by several CASE tools that moreover conform a specific architecture. For instance, the user can select an object and generate the classes necessary to make it conform to a Design Pattern, make an executable component out of it or convert it to the class that can be accessed remotely in accordance with the Java RMI specification. In this context, the designer usually abstracts from the communication paradigm that underlies the connection (i.e., the bus) between model and simulator.

Automatic code generation from a model is also termed forward engineering. Sometimes changes or even development are undertaken directly on the program code. There are tools that support reverse engineering, they e.g. attempt to automatically generate class and sequence diagrams from source code. A CASE tool supporting both forward and reverse engineering is said to provide a means for round-trip engineering. Simultaneous round-trip engineering eliminates code and model mismatch; changes to the model or the code can be made and both stay consistent with each other, no extra steps are needed.[1] A further concept is that of re-engineering, which denominates the task of changing an existing system in order to meet new requirements.

Also, as an important feature in order to increse the level of automation in the development process, many CASE tools provide facilities for reporting, e.g. for analysis and documentation.

[1] This is true with respect to structure. Regarding behavior, on the contrary, some adjustments have to be done by hand. In the UML realm, code can be generated from both state diagrams and activity diagrams, sequence diagrams can be generated from code and state diagrams from sequence diagrams. All these functionalities composed, on the one hand, do not suffice for real mismatch elimination, and on the other, to our knowledge they are not integrated in a single CASE tool.

3.2 Current solutions

The present and most well-known CASE tools and the support provided by them can be characterized as detailed below. We restrict the description to representative commercial tools. This summary is not intended to be complete: it provides a survey on current CASE tool *concepts* and refers to representative tool implementations.

UML

The Unified Modeling Language (UML [21, 23, 22]) is a collection of notations used to describe a software intensive system in its different stages of development as well as from various perspectives. It is an object-oriented notation and meanwhile has become a de facto standard. UML defines use cases, class, object, sequence, collaboration, activity, state, component, and deployment diagrams, and puts the Object Constraint Language (OCL [35]) at disposal for the specification of additional conditions that cannot be diagrammatically stated and are to be applied against a modeling element. In other words, UML has been designed as a universal notation for the whole development process from requirements analysis to implementation, at least for business applications (although many efforts are being undertaken in order to make UML applicable in other fields).

However, the application of the different diagram types as well as the relationship of them with each other is not clearly enough stated. Moreover, the semantics of UML diagrams is not univocally defined. Thus, according to the numerous different interpretations of behavior models, results in analysis for "seemingly equivalent models" differ widely. Furthermore UML lacks a component concept that could serve as basis for software architecture; see [12, 11].

As a result of these ambiguities, UML is so far mostly used only for documentation purposes, unfortunately.

UML-RT

The UML and powerful modeling constructs originally developed for the modeling of complex real-time systems in the Real-Time Object-Oriented Modeling language (ROOM [27]), have been combined into UML for Real-Time (UML-RT [26, 18]). It is directed at developers of complex real-time software systems (e.g., telecommunications, aerospace, defense, and automatic control applications).

The complexity of real-time systems arises from aspects like concurrency, dynamic behavior, variable loading, and memory and processing limitations on target platforms. In the continuously expanding area of telecommunications, and because the business gets more and more competitive, to those aspects we may add the necessity of a faster and cheaper development process of real-time applications.

The execution model of UML-RT focusses on high-level system modeling and leaves open many implementation concerns like e.g. performance and efficient code generation.

ASCET-SD, Stateflow, and Betterstate

The tools ASCET-SD [2], Stateflow [8] and Betterstate [4] facilitate the development of time discrete embedded systems by means of structure diagrams and a variant of statecharts. Systems are specified at a rather detailed level of abstraction: communication between components is specified by means of shared variables, there are no abstractions which ensure the reception of sent messages or avoid duplicate reading. Behavior is modeled at an implementation-oriented level including implementations of algorithms (in the programming language C) and the scheduling of the computation of components.

Betterstate and especially ASCET provide facilities for efficient code generation and simulation. For Stateflow, the external code generators Target Link [6] and Beacon [3] are availavle. Support in all these tools is mostly limited to sysetms with one electronic control unit (ECU). Though, limited Modeling extensions for distributed systems exist for Stateflow, and a as more general solution, the toolset TITUS [] is under development but not available yet.

Matlab/Simulink and MartixX

Matlab [8] and MartixX [] enable the specification of both message and time discrete and continuous embedded systems by means of block diagrams from control theory. The system structure is specified in terms of control blocks and signal/information flow. The behavior of each control block is specified by a control algorithm. Similar to discrete, state based descriptions, the specification includes the scheduling of the execution of control algorithms: They can be activated periodically (specified by sampling rates), continuously or triggered by events.

Both approaches provide a simulation environment which allow to analyze the behavior of a control system in "logical time". This simulation abstracts from actual consumption of resources by control blocks, and thus of schedulability problems. The simulation is computed in discrete steps. The granularity of this discretization can be specified in order to constrain the divergences between simulation and actual physical behavior to an appropriate measure.

Both tools include the integration of state based description techniques for discrete system parts: Matlab is interconnected with Stateflow and MatrixX with Betterstate.

VCC

VCC [5] provides a rather detailed abstraction level in the specification of embedded systems which focuses on the analysis of run time behavior and performance aspects in the context of concrete hardware. VCC provides hierarchical structure diagrams with components, connectors and ports similar to ASCET-SD or UML-RT. Component behavior is specified by a variant of C programs.

VCC provides a mapping of the logical component structure to a hardware model which includes the specification of clock cycle times of processors, the consumption of clock cycles by algorithms and characteristics of communication busses. Based

on both models, a simulation environment enables both the logical analysis of the systems functionality and performance analysis considering resource consummation and timed behavior.

AutoFocus and Esterel

Verification support by the tools mentioned above is mainly restricted to simulation environments. AUTOFOCUS [**?**, **?**] and Esterel [7] tools provide a more elaborated verification support:

AUTOFOCUS specifies the structure of embedded systems by means of system structure diagrams (SSDs) consisting of components, port interfaces and communication channels, and the logical behavior by a syntactically restricted variant of Statecharts (STDs). Based on a formal semantics for these techniques, AUTOFOCUS provides support for the generation of test cases (in most other CASE tools only, if at all, the execution of tests is supported) and model checking techniques, as well as a simulation environment and code generators.

Esterel specifies embedded systems by means of a synchronous, formally founded programming language and provides facilities for verification using model checking and testing, and test case and code generation (also state based graphical techniques: Esterel studio)

Doors

The CASE tool doors focuses on the traceability aspect of requirements engineering. Traceability is supported by linking interdependent artifacts in system development specified in general purpose tools such as text editors and spreadsheets and special purpose CASE tools through a hyper link structure.

This way, doors supports the navigation through artifacts of development. Further, it provides facilities for automated generation of documentation. The feasibility of doors' link structures depends on two aspects: first, the granularity at which artifacts of special purpose CASE tools can be linked (this depends on interfaces of those tools). Second, and most important, a clear model of requirements and their relations. is necessary. This has to be provided by the user.

Others

As stated above, this survey is far from being complete. It focuses mainly on design-oriented approaches which have been used or assessed in automotive development projects. Beside them, there are a lot of other approaches:

These comprise tools based on Petri nets such as PEP [], tools successfully used in related domains such as SDL (e.g. Tau [**?**]) in telecommunication, and numerous academic languages and tools focusing on specific aspects such as timed behavior (*Timed Automata* [**?**, **?**] and *Giotto* [**?**]), interface specification (*Interface Automata* [**?**]) and hybrid systems (e.g. [**?**, **?**]). Also there are approaches for meta CASE tools (e.g. METACASE []) which provide facilities to customize general notions like state based de-

scriptions to application specific needs.

3.3 Strengths and weaknesses

The above addressed CASE tools show strengths as well as weaknesses. This section is devoted to identify these.

Comprehensiveness and abstraction

A principal challenge to CASE tools in order for mastering complex development projects is the support of appropriate, comprehensive abstractions for each development task and the integration of different abstraction levels.

By far most of the CASE tools discussed above provide graphical description techniques both for the specification of a system's structure and behavior. The two dimensional presentation eases comprehension of complex structures and behaviors compared with the linear structure of programming languages, since artifacts like control loops have a closed, intuitive representation. Graphical modeling languages are no universal remedy for comprehensive models: As abuse of programming languages can result in spaghetti code, abuse of the expressiveness of modeling techniques can result in, say, ravioli structures.

Whereas modeling languages cannot be blamed for the possibility to abuse them in general, there are weaknesses concerning their structuring mechanisms which prolifer poorly structured modeling: For example, the UML does not enforce restrictions on the entering and exiting substructures of statecharts, and the relation between state based behavior and algorithmical and data aspects is often ill-fated.

Assessing today's CASE tools concerning their support for abstractions requires considering individual aspects: some CASE tools provide abstraction from concrete communication specifics such as to ensure message transfer. For example, the communication model guarantees that every message sent to a receiver is actually being read. ROOM also allows for the abstraction of actual execution time of specific interactions. Case tools like ASCET-SD and VCC enforce a much more detailed modeling because they aim at the analysis of timed and communication aspects. Still, languages like ROOM have severe shortcomings concerning abstraction: computation - and even communication constructs - need to be specified at the level of implementation in programming languages, and there exists, for example, no abstract construct for synchronization tasks.

These examples reveal another serious problem: Each development phase has its own needs for abstraction, appropriate abstractions for requirements analysis fundamentally differ from design abstractions and implementation. The discussion of state of the art CASE tools in the previous section shows that the level of abstraction in those tools is manifested by language design decisions. For example, ROOM's communication paradigm is inadequate for time and performance modeling, whereas the paradigm of ASCET-SD, VCC and others enforces over-specification if applied in

early phases of development.

Integrations of CASE tools in order to cover different development phases are mostly not available yet. Only hyperlink structures (without semantical foundation, see below) through the doors tool and a few linkages between modeling tools and code generators and validation tools are available so far. Beside that, integration of CASE tools is still a matter of research, e.g. in [**?**, **?**].

Consistency of models

As stated in Section 3.1, consistency between views can be reached by explicit conditions or else based on an integrated meta model. In general, the quality of mechanisms ensuring consistency depends on the completeness of consistency conditions or, in the case of meta model based tools, the completeness of the meta model and the extent to which it is free of redundancy.

Consistency in current case tools is, if at all, preserved only partially so far: For example, many tools allow the specification statecharts with transitions not connected to states or communication channels not connected to components. However, also strict policies in order to preserve consistency can have their disadvantages: systems development usually considers a rough sketch of the system an its requirements first, bookkeeping mentality can severely hinder creativity. As an example, some CASE tools require to have all datatypes already predefined when starting the specification of an interaction scenarios by a MSC, i.e. force to specify details before the rough lines of the considered protocol.

These examples show that there is a need for improving CASE tools both by more elaborated consistency check mechanisms and in a way that the control of the process of ensuring it is kept by the user.

Validation and code generation

Many of the CASE tools mentioned above support validation usually by way of animation (also called simulation and model-level debugging). However, in most cases simulation is only supported at a very detailed abstraction level. Simulation of dynamic models is a tool for testing the behavior of design in advance, before spending the time and budget on the cycle of coding-testing-coding. Higher quality in shorter time is gained thanks to the ability to verify and validate design very early at the development life cycle. For instance, multi-threaded objects are able to be checked for deadlocks.

Simulation is especially feasible for deterministic models; for non-deterministic ones, it is not clear if a simulation run covers all relevant behaviors. Thus in extreme cases, the actual system may behave completely different from the behavior shown by the simulator.

Use cases, sequence diagrams, statecharts or Petri nets can be animated. Some CASE tools, e.g. SoftModeler [17] and ARTiSAN Real-time Studio [33], allow step by step execution and animation of multi-threaded sequence diagrams. We find CASE tools,

e.g. ARTiSAN Real-time Studio and Rhapsody [15], that support statechart animation, in some cases both forwards as well as backwards. The user can interact with the animation and even alter it. During animation, the actual state of the object can be explored, and event occurrences can be executed, state transitions are visualized. Petri net animation is available only in academia; see [19].

Whereas there are many examples for successful applications of simulation and code generation for testing and rapid prototyping, code generators often fail to fulfill requirements of specific domains to an extent that generated code could be used as the actual implementation. Examples relevant for the automotive domain are the need for sophisticated optimizations concerning code size and execution time.

A more formal approach to validation are the mechanisms put at disposal by tools like e.g. AutoFocus [14], which integrates both a model checker and a theorem prover, that support mathematically founded simulation and debugging of models. Also Esterel Studio is a graphical modeling environment that offers formal verification facilities and automatic generation of code, which supports interactive behavior simulation.

Traceability

Traceability in a development project is viewed as the problem of maintaining an information system that keeps the relevant links between artifacts developed and delivered by a development process, in particular implementation and dependency relationships in the model. Traceability facilitates user requirements testing as well as realization of requirements changes. There are CASE tools, e.g. doors/ERS, MagicDraw (see [28]) and Enterprise Architect (see [31]), that let the designer capture this information using realization links. Analysis diagrams associated with processes, use cases, classes, etc., capture the realization relationships.

So far, a formal definition of how to specify a traceability relation between model elements of different views or diagrams (be this within the same tool or spanning across different CASE tools) and of its formal semantics is missing. A proposal to overcome such difficulties with stress on change of requirements can be found in [34]; the document structure defined therein is currently used in a research project with DaimlerChrysler.

Reuse

Concerning the efficiency of systems development, a frequently uttered request is to extend the potential of reusing artifacts, components and code. Most description techniques used in state of the art CASE tools are labeled as object oriented, and one of the major goals of object orientation was to improve the potential of reuse. Nevertheless reuse is poorly supported in CASE tools:

Most case tools provide import and export functionality for parts of models. However, more elaborated mechanisms supporting e.g. renaming, instantiation, parameterization and adaptation of model elements (e.g. class diagrams enable the extension of classes by subclasses, but what about the behavior?) are almost entirely missing so

far.

The necessity for more powerful solutions for reuse is witnessed for example by a successful idea - design patterns: Although there is a fast growing literature on proposed patterns in design, there is virtually no commonly used case tool for which a library of patterns or standard model element is available[2].

Real-time requirements

CASE tools are available, for instance Rapid RMA (see [29]), that contain multiple analysis capabilities, which allow designers to test software models against various design scenarios and evaluate how different implementations might optimize the performance of their systems. In this way, potential scheduling bottlenecks in both soft and hard real-time systems can be isolated. For instance Together ControlCenter is able to use an integrated simulator for performance analysis of real-time systems or to export to XML and use an external simulator.

Independence of tool vendors

The most notable approach to limit the mentioned threats by dependence of developers on single tool vendors is standardization. Concerning CASE tools, the most relevant effort in this context is standardization of specification languages and their representation.

The discussion of existing tools above shows that many specification languages in current CASE tools are neither standardized nor conceptually integrated so far. There are only a few import/export functionalities which allow to transfer models between competing tools. Experiences from related fields like computer aided design (CAD) suggest that we cannot expect much more than, at best, peer to peer exchange: Numerous efforts have been made to establish feasible exchange formats for tools for computer aided design but, until now, all approaches face the problem that model exchange leads to severe loss of information.

Still there are specification language standards relevant for the automotive domain, namely ITU MSC and SDL, and the UML. Since the UML is supported by numerous tool vendors, especially for this language the exchange of models between tools from different vendors became quite relevant. The most recent approach by the OMG is the Meta Object Facility (MOF) [] as meta model for object oriented design and the XML based representation standard XMI [] for MOF compliant models. Further relevant exchange standards independent of the UML are CDIF [] and MSR [].

Most UML CASE tools claim to support XMI. However, the feasibility of model exchange based on XMI is still rather limited until now: Due to weaknesses in the specification of both the UML and XMI and because of the ongoing development of both stanards (partly without downward compability) model exchange works at best for subsets of models. Experiences show that even the exchange between tools from the

[2] Well, this is a general remark on case tools. Actually, for embedded systems the literature on patterns is rather poor ...

same vendor often leads to loss of information.

With the UML, the XML based exchange format XMI is defined. Due to continuing changes in the meta model and weaknesses in the XMI specification, diagram interchange between different CASE tools is not possible so far, even between CASE tool versions of the same tool vendor.

4 Conclusion

While CASE tools for automotive applications have been used with partial success—notably for rapid prototyping—they are not suited for an integrated application throughout the software development process.

We believe that the main reason for this failure are the lack of suitable domain-specific abstractions and description techniques for automotive software artefacts, the problems with traceability of these artefacts through the development process, and the open question of certification of the controller software. In particular the lack of suitable abstractions, descriptions and semantics impairs the integration of heterogenous tools by different vendors that is needed for flexible development processes and a sustainable market for both tool vendors and tool customers.

That flexible tool integration and commercially viable markets for tools are indeed possible, however, can be seen from the example of hardware design tools: Here common syntactical and semantical concepts allow the integration of tools for different abstraction levels (analog level, register-transfer level, function block level, ASICs, etc.), for different purposes (design, simulation, verification, power and chip area estimation) and different vendors. Unfortunately, we do not have an answer to the question of how a similar state of affairs might be reached for automotive CASE tools.

Acknowledgment.

References

[1] Common object request broker (corba).

[2] Ascet-sd product info. 2002.

[3] Beacon for simulink/stateflow. 2002.

[4] Betterstate product info. 2002.

[5] Cadence vcc product information. 2002.

[6] dspacs products. 2002.

[7] Esterel. 2002.

[8] The mathworks product family. 2002.

[9] A. Blotz, F. Huber, H. Lötzbeyer, A. Pretschner, O. Slotosch, and H.-P. Zängerl. Model-based software engineering and Ada: Synergy for the development of safety-critical systems. In *Proc. Ada Deutschland Tagung,*, March 2002. To be published.

[10] Peter Braun, Michael von der Beeck, Martin Rappl, and Christian Schroeder. Automotive Software Development: A Model Based Approach. In *In-Vehicle Software*, SAE Technical Paper Series: 2002-01-0875. SAE, 2002.

[11] Manfred Broy and Johannes Siedersleben. Objektorientierte Programmierung & Softwareentwicklung: Eine kritische Einschätzung. *Informatik Spektrum*, pages 3–11, February 2002.

[12] Manfred Broy, Michael von der Beeck, Peter Braun, and Martin Rappl. A fundamental critique of the UML for the specification of embedded systems. 2001.

[13] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: A time-triggered language for embedded programming. In *Proceedings of EMSOFT 2001, LNCS 2211*, 2001.

[14] F. Huber, B. Schätz, A. Schmidt, and K. Spies. Autofocus—a tool for distributed systems specification. In *FTRTFT'96, LNCS 1135*, 1996.

[15] I-Logix. Rhapsody. http://www.ilogix.com/products/rhapsody/index.cfm.

[16] Edward A. Lee. Embedded software. In M. Zelkowitz, editor, *Advances in Computers*, volume 56. Academic Press, London, 2002. To appear.

[17] Softera Ltd. Softmodeler. http://www.softera.com/products.htm, 2002.

[18] A. Lyons. UML for Real-Time Overview. *Objectime Ltd.*, April 1998. `http://www.objectime.com/otl/technical/umlrt.html` .

[19] University of Oldenburg Parallel Systems Group. Pep. http://parsys.informatik.uni-oldenburg.de/p̃ep/, 2001.

[20] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[21] Rational. UML Notation guide, version 1.4. 2001.

[22] Rational. UML Semantics, version 1.4. 2001.

[23] Rational. UML Summary, version 1.4. 2001.

[24] Christine Rosette. Elektronisch gesteuerte Systeme legen weiterhin zu. *Elektronik AUTOMOTIVE*, pages 22–23, 2002.

[25] B. Schätz, A. Pretschner, F. Huber, and J. Philipps. Model-based development. Technical Report TUM-I0204, Institut für Informatik, Technische Universität München, 2002.

[26] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. Available under http://www.objectime.com/uml, April 1998.

[27] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. Wiley Professional Computing. John Wiley & Sons, 1994.

[28] KAPITEC Software. Magicdraw. http://www.kapitec.com/Produits/MagicDraw/en/presentation.htm, 2001–2002.

[29] Tri-Pacific Software. Rapid rma. http://www.tripac.com/html/prod-fact-rrm.html, 1999.

[30] R. Soley. Model Driven Architecture. OMG white paper, 2000.

[31] Sparx Systems. Enterprisearchitect. http://www.sparxsystems.com.au/, 2000–2002.

[32] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1999.

[33] ARTiSAN Software Tools. Artisan real-time studio. http://www.artisansw.com/products/professional_overview.asp, 2001.

[34] Antje von Knethen. *Change-oriented requirements traceability support for evolution of embedded systems*. PhD thesis, Universität Kaiserslautern, Germany, 2001.

[35] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Object Technology Series. Addison-Wesley, 1998.