

Modeling Embedded Software: State of the Art and Beyond

Bernhard Schätz, Jan Romberg, Oscar Slotosch, Martin Strecker, Alexander Wisspeintner, Tobias Hain, Wolfgang Prenninger, Martin Rappl, Katharina Spies

Technische Universität München, Fakultät für Informatik
Boltzmannstr.3, D-85748 Garching (Munich)

{schaetz,romberg,slotosch,strecker,m,wisspein,hain,prenning,rappl,spiesk}@in.tum.de

Abstract

State-of-the-art software development increasingly relies on describing the system (or software) graphically, abstracting from an actual implementation platform, and supporting to generate an executable system out of the model. Similarly, in electrical engineering often graphically represented models are used to describe the controlled system including its low-level control behavior. Since developing embedded software requires using abstract, functional models of the software as well as incorporating aspects from the implemented algorithmic behavior, a suitable model-based development process must integrate models from both fields.

By comparing the results from modeling an automotive embedded controller software module with eight different state-of-the-art CASE tools, we show what kind of abstractions, views, and models have proven useful in that domain. Furthermore, we show what analytical and generative development steps are currently supported by CASE tools. Based on those experiences and results from other research on efficiency and effectiveness of software engineering techniques and CASE support, we derive the properties of a model-based development process and illustrate it by example support functionalities.

Keywords: Embedded systems, CASE, tool support, model-based, views, model analysis, model generation

1 Model-based Development: State of the Art

When looking for a definition of what model-based software development is (or is not), there is a wide range of different interpretations. However, most approaches have in common:

- the use of graphical representations for the system under development
- the possibility to describe the system (or software) with a certain degree of abstraction from the actual implementation platform
- the possibility to generate an executable system out of the model

To assess the state of the art, we compare the results of applying eight different tools for the development of embedded software to a common problem: the specification of a software module controlling comfort electronic functionality. The applied tools are

- ARTiSAN RealTime Studio by Artisan Software
- ASCET-SD by ETAS GmbH & Co.KG
- AutoFOCUS by Technische Universität München
- MATLAB/StateFlow by The MathWorks Inc.
- Rose RealTime by Rational
- Rhapsody in MicroC by I-Logix Inc.
- Telelogic Tau by Telelogic Inc.
- Trice Tool by Protos Software GmbH

With each tool, a model of a controller software module was developed, based on a given textual requirement specification. The requirement specification of the controller was taken from a revised version of a controller specification provided by F. Houdek, Daimler Chrysler AG.

Besides providing the above-mentioned support, those state-of-the art CASE tools for embedded systems additionally offer support

- to describe the system using different hierarchic views
- to use message- or signal-based communication in the operational model
- to include timing aspects in the description of the system
- to check the model for inconsistencies, mainly on the syntactic level (e.g., undefined identifiers, type mismatches)
- to simulate the system at the level of the description
- to deploy the generated code to a host and a target system.

In Section 2, we give an overview over the results obtained when assessing the above tools – an extended version can be found in [SHH+03]. In Section 3 we discuss what possible improvements to CASE support should be added to arrive at a model-based development process.

2 Assessing the State of the Art

Since they target different application domains, the discussed tools focus on different aspects of the development process, e.g., modeling the system under development, supporting the test of the system, or generating and deploying code to the embedded controller. Therefore, to simplify a comparison of the different functionalities offered by the tools, the tools were analyzed according to different criteria using a common questionnaire. The following subsections give a short overview over the results.

2.1 Modeling the System

Generally, for an embedded system, the interactive behavior of the system or its components plays a more important role than the complexity of its data structures. Therefore, a central aspect of a development tool for embedded systems is the modeling these interactions to treat them at a higher level of abstraction than, e.g., in terms of method calls between objects. In general, all of the selected tools model an (embedded) system as a collection of individual reactive components, communicating by some form of message or signal exchange. The environment is accessed by receiving messages from sensors and sending messages to actors. Each component has an associated behavior describing its input/output relation in form of internal computational data flow or state-machine; the behavior is triggered by the reception of a message/signal or some timing event. Time-treatment (access to clocks or use of timers producing timeout events) is available to deal with (weak) real-time constraints.

They differ, however, concerning the level of abstraction from the implementation they use. Some tools

rather consequently use this abstract model (e.g., AF, Matlab, Rhapsody, Tau, Trice); others at least partly keep the implementation view, modeling components, e.g., by object communicating by method call (e.g., ARTISAN). This is reflected in the description techniques as well as the operational model supported by the tools.

2.1.1 Description Techniques

All tools support four common classes of description techniques:

- **Structural Descriptions** describing the architecture of the system consisting of components, interfaces (e.g., ports, sensors, actors), communication paths (e.g., channels, connections). Examples are System Architecture Diagrams in Artisan, or Block Diagrams in ASCET.
- **State-Based Descriptions** describing the behavior of the system using states, transitions, actions or events, and timing annotations. Examples are State Transition Diagrams in AF and Matlab (different variants).
- **Scenario-Based Descriptions** describing exemplary execution sequences consisting of interactions between components or their continuous data flow (e.g., in an Oscilloscope-like manner). Examples are Sequence Diagrams in Rhapsody and Rose.
- **Data Description** describing the data types used to define messages, signals, or variables. Examples are Class Diagrams in Tau and Trice.

The first three are generally defined using a graphical description (including complex textual expression, e.g., for the description of transitions or events/interactions), the latter either graphically or textually. Furthermore, some tools offer additional description techniques like the schedules of activation, UML Use Cases, Activity Diagrams, or Package Diagrams. To structure the specifications, throughout the tools, each (graphical) description technique supports hierarchical structuring (e.g., component/subcomponents, state/sub-states).

In general only those description techniques were intensively applied which are integrated in the development process (see also 3.3): either because a simulatable specification or code can be generated from them, or because they are generated from other descriptions (like interaction descriptions). Since the case studies are focusing on modeling the system rather than implementing it, the above-mentioned aspects of defining task and schedules as well as organizing the system in packages were kept to a minimum.

In general, the tools focus on the four common description. Additionally, were available, use cases were used in an early stage to structure the functionality and to collect interface information. Since basically, structure, behavior, and data descriptions are sufficient to describe an executable model of the system, all sets of description techniques offered by the tools were considered to be sufficient. If no interaction descriptions by event-based forms like sequence diagrams were available (e.g. Matlab), those were found missing.

The use of hierarchic graphical description techniques was considered to greatly improve the clearness of the model. While dependent on hierarchy support, operational model, and level of abstraction, the complexity of the specifications is more or less within the same order of magnitude for the tools. Especially, the average complexity per view (diagram) is similar between most tools (4 nodes, 7 edges); this indicates suitable modular description techniques supporting readable designs:

- **Views:** The number of views varies between 14 and 90, with an average around 40 views used to model the controller.
- **Nodes:** A total number of about 100 nodes average where needed to model the system.; the average number of nodes per view is about 4.
- **Edges:** The total number of edges used in the specification ranges about between 150 and 300 edges; the average number of edges per view is about 7.
- **Visible annotations:** The average length of visible annotations used to specify the controller is between 30 and 60.

2.1.2 Operational Model

Besides complexity, the operational model also influences the expressibility/ease of use of the modeling language. When describing the behavior of a reactive system, the description of interactions between its components plays a central role. Two different forms of synchronization influence the interaction:

- **Message Synchronization:** This form describes the coupling of sender and receiver of a communication.
Message-synchronous communication corresponds to a handshake communication blocking the sender of a message until the receiver accepts the message. With *message asynchronous* communication, the receiver of the message does not influence the sender of a message; the receiver will always accept the message. This is either the case with *buffered communication* where the receiver buffers unread message until consumption or with *signal-based communication* where unread messages are overwritten by new arriving messages.
- **Time Synchronization:** This form described the synchronization of the actions of different components. In *event-driven* systems, behavior is triggered by the occurrence of events; in *time-driven* systems, behavior is triggered by the passing of time.

The applied tools use different combinations of those synchronization aspects. Generally, the users of the tools have found all operational models sufficient. However, extensions of the models with simple properties can sometimes reduce the ease of handling (e.g, explicit buffering of messages in signal-based communication).

Concerning support for the real-time aspects of embedded systems, there are different possible approaches:

- **Timed model:** The operational model explicitly deals with time. Depending on the synchronization mechanism of the model, there are different variants: *event driven* models generally use some form of *timeout* event triggering behavior; *time-driven* models usually have a scheduled execution model supporting access to some system clock variable.
- **Timed deployment:** The operational model does not directly deal with time. At most, timing conditions can be added as annotations.

Different timing models are used in the tools. Independent of the timing model, to meet real-time bounds, during deployment schedules and frequencies depending on the hardware platform have to be defined or connections to the real-time features of the operating systems must be established.

2.2 Development Process

Since – as argued in Section 3– a major advantage of a model-based development process lies in the analysis and construction support on the level of the abstract model, we take a close look at that subject (especially in Subsection 2.2.2).

2.2.1 Applied Process

Obviously, the applied process is defined by the description techniques and process support supplied by the used tools. Nonetheless, a common development process independent from the tools can be established:

- **Analysis:** The purpose of this phase is to get a better understanding of the system and to identify groups of functionality. Due to restriction of notations, this phase is more explicit in tools supporting additional analysis description techniques like UML use cases. Nevertheless, in all tools Step 2 was performed:
 1. Coarsely structuring the functionality of the system, using use case-like diagrams
 2. Defining the system boundary (e.g., actors, sensors, busses), using structural diagrams
 3. Exemplarily defining the functionality of the system using scenario based diagrams
- **Design:** During this phase, the system was generally partitioned to support concurrent engineering and modular development. Note that this partitioning is performed on structural decomposition, but based on the functional structuring identified in the Analysis phase. These steps were generally explicitly performed in all tool applications:
 1. Defining and refining the system structure by introducing (sub-)components, using structural diagrams
 2. Adding behavior to the (sub-)components, using state-based diagrams or data-flow diagrams
 3. Validating and refining the behavior of (sub-)components, using simulation
- **Integration/Validation:** In this phase, the different components were integrated into one system (generally, due to the component model supported by the tools this requires no additional steps) and validated by simulation. Generally, Step 1 step was performed by all tool applications:
 1. Generating simulations of the complete system, either generating scenario-based descriptions of simulation runs or using simulation interfaces
 2. Comparing the simulation results with the scenario-based descriptions

2.2.2 Applied Process Support

Development Operations

Generally all tools support simple development operations (e.g., cut-and-past of elements within one diagram). Complex development operations (i.e. supporting complex stereotypic or application domain specific operations) generally focus on the support of the implementation. Typical examples are the generation of schedules for computations (e.g., ASCET, ROSE, Tau) or optimizations for certain platforms. Depending on how strong a tool supports the implementation phase, some of these functionalities may not be available if the tool focuses on analysis and design (e.g., ARTiSAN). Generally, however, most of the tools only have restricted complex development support for the analysis or design phase (see also Subsection 3.3.2).

Reverse engineering generally focuses on the structural parts of a system (e.g., generation of class diagrams) or is limited to code generated by the tool and hardly modified.

Consistency Ensurance Mechanisms

When building a large specification broken up into several modules or diagrams, frequently errors arise at the interfaces between those parts. Additionally, specification errors also arise within each module or diagram when parts of these specifications are missing or to not fit together. CASE-tools can help to detect those inconsistencies at different levels:

- **Code Level:** On this level, the tool does not directly support model consistency. Rather, syntactic consistency is only defined on the level of the generated code; generally, the tool however can related code-level errors back to the model (e.g., by relating a compiler error to the relevant element of the model).

- **Syntactic Diagram Level:** On this level, the tool makes sure that a single diagram (or view) is well-formed. This is the most basic form of model consistency (e.g., well-formedness of trigger expression)
- **Syntactic Model Level:** On this level, the tool ensures syntactic consistency between diagrams referencing the same elements of the model (e.g., messages used in an interaction description/Sequence Diagram are defined according to the type of the channel as defined in the structural description/Architecture Diagram).
- **Semantic Diagram Level:** On this level, the tool ensures well-definedness of a single diagram, usually requiring some form of execution or verification (e.g., no two transitions from one state can be enabled by the same triggering events).
- **Semantic Model Level:** On this level, the tool ensures that the dynamic aspects of the description are consistent - generally, this requires some form of (symbolic) execution or verification (e.g., consistency between State Diagram and Sequence Diagram).

Syntactic diagram level consistency can either be ensured constructively (e.g., structural editors supporting only the construction of a connecting channel between ports of corresponding types) or by analysis (e.g., editors checking well-formedness of trigger conditions). Syntactic model level consistency, too, can either be ensured constructively or by analysis. Tools do generally not support semantic diagram consistency. Few exceptions exist, e.g., the non-determinism check (AF). Finally, semantic model consistency in most tools only supports a constructive approach. Generally, this is limited to the construction of scenario based interaction descriptions, e.g., in form of sequence diagrams; most tools support this. Few tools support the verification of such an interaction description against the state-based behavioral description, e.g. by checking whether a sequence diagram can be executed by the state machines of the corresponding components.

2.2.3 Quality Management

The tools show only slight differences in their support for quality management:

- **Simulation Support.** The tools generally offer (host) simulation of the model. The tools differ as to which kinds of displays are available. Some tools do not generate target code, so no target code simulation is offered. Even if direct target simulation facilities are not provided, as in AF, the generated target code can in principle be simulated on the target hardware using tools of third-party vendors.
- **Testing.** Almost all tools allow parameter vectors to be played back during a simulation, and current simulation data to be saved for later use, for example in regression testing. Apart from that, there is little support for test data management, like batch processing of test suites and generation of test statistics. Test coverage analysis is provided by Rose (interface to an external tool) and Matlab. Automatic generation of test cases out of the model is to some extent offered by AF.

In a nutshell, the most important quality assurance method is simulation. The tools are generally very well developed in this area. However, “debugging” and “testing” are just understood as variations of simulation. Genuine support for testing (automatic derivation of test cases; test analysis including test coverage metrics) is not provided by the majority of tools, static analysis techniques are missing almost entirely.

2.2.4 Incremental Development

Support for reusability and the restriction of changes to localized parts of the model were generally found to be adequate. All of the evaluated tools supported some notion of a reusable component or actor, thus simplifying architectural changes and reuse of existing components. As the only research tool in the comparison, AF was found to lack support for bottom-up structuring. Moving from the first iteration to the second, changes in the specification were found to be restricted to some components, not affecting the specification as a whole. Reuse of partial specifications was typically performed by simple copy/paste mechanisms; library mechanisms as in Matlab and AF have also proven to be helpful.

As of now, most of the tested tools lack automated support for restructuring and refactoring. As a notable exception, the two UML-RT tools, Rose and Trice, offer automated composition and decomposition of actors (“aggregate function”).

3 Model-based Development: Picture of the Future

The construction of reliable (embedded) software is of course possible without an explicit model-based tool-supported development process. However, research results suggest that such a process can contribute essentially to increased the quality of the developed product as well as to an increased efficiency of the development itself:

- [Jon91] shows that more than 50 % of serious errors are made during design (25 % during implementation); about 30 % of medium class errors are made during design (30 % during implementation)
- [Jon91] shows that analytical techniques performed on early-phase description of the product (e.g., structured approaches, design reviews) require generally at least less than 50 % of the effort in both error detection and correction needed for later-phase techniques (e.g., integration test, field test)

- [Jon91] shows that those analytical techniques of the early phases are at least twice as effective to detect errors of the early phases than those later-phase techniques.
- [BMJ+96] shows that especially in a development process requiring a high level of product quality, CASE support can significantly increase productivity.

Therefore, in order to increasing both the *efficiency of the development process* (by increasing the degree of mechanization), and the *quality of the development product* (by decreasing the amount of possible defects), however, support for a model-based development for embedded systems should exceed the following forms of CASE tool support:

- **Using an implementation-level model:** Modeling the system at implementation level rather than at a more abstract level leads to a limited development process, focusing on the implementation and integration phase. Thus, e.g., defect analysis is limited to implementation level defects. This excludes simple defects like message interface incompatibility between processes executed on different nodes since those messages are described as a byte-block oriented bus-protocol. Furthermore, due to the gap between the earlier phases and the implementation level, even design specifications are not always related to the implementation. This is often leading to inconsistencies between the design and the implementation.
- **Using an OO-model for embedded software:** OO-based approaches supply different views of the system including non-executable views supporting early phases (e.g., interaction scenarios) making consistency analysis available in those phases. However, those views offer only limited abstraction from the OO operational model; e.g., method calls are used to model interaction between tasks rather than the more suitable message-/signal-based communication. Furthermore, additional domain-specific aspects (e.g., bus schedules, task switching) are not modeled explicit. Therefore those aspects are laid off to the coding phase outside the modeling capability of the tool, leading to similar problems as with implementation-level models.
- **Using a Draw-and-Generate Tool:** Domain-specific approaches generally support a more detailed model including aspects like preemption or time-driven communication allocatable to bus slots; the tools make use of this information to generate deployable code. However, sophisticated analysis techniques are not available on the level of the model; therefore, the analysis of defects introduced on the level of the model is performed manually or is delayed until the execution of the generated model. Thus, e.g., the tool does not ensure the consistency between time-driven communication and allocated bus schedules.
- **Using a loosely coupled tool chain:** A loosely coupled tool chain generally splits the development process in tool-related phases with substantial gaps between those phases; e.g., scenario-based descriptions of the behavior that cannot be used to generate equivalent test cases on the implementation level. Therefore, the tool chain depends heavily on those forward and backward integrations and the common model bridging the tool chain and thus limiting the degree of coupling. If, e.g., the bus signal communicated on the implementation level cannot be related back to messages communicated between abstract components, counter-examples cannot be expressed on that abstract level.

In contrast, for optimal support, a model-based software development process must fulfill the following properties:

- **Adequate Models:** During the development process, different aspects of the system under development must be addressed (for the domain of embedded systems, e.g., partitioning and deployment, scheduling) during different phases. Specific models are available for the different phases (e.g., requirements analysis, design, implementation, integration) of the development process. Since software systems, and especially embedded software systems, are moving from monolithic single-functionality programs to distributed, interactive multi-functionality networks, a central aspect of these models is treatment of interaction and communication as well as time-related aspects. Furthermore, models must support specific aspects needed for the application domain (for the domain of embedded real-time systems, e.g., notions like messages/signals, task, schedule, controller, bus). By supporting domain- and application specific modeling elements, model information about the system under development becomes available, leading to stronger analysis and generation techniques (e.g. in the domain of embedded systems, checking the worst case time bounds of a task, or generating a bus schedule from the communication description of a component model).
- **Abstract Models:** Models should contain only those aspects needed to support the development phase they are applied for (e.g., modeling the interaction between components by messages rather than method calls to the bus driver or the operating system). By abstraction models reduce the complexity of description as much as possible as well as the possibility to produce faulty descriptions (e.g., ensuring type correctness between communication ports rather than using byte block messages on the level of bus communication). Furthermore, abstract models also support descriptions of the system under development in early analysis and design phases, making analysis and generation support available even at early stages of the development process (e.g., completion of state-based description of a component to introduce standard behavior for undefined situations).
- **Integration by Analysis:** The relation between different models during the development is supported. This includes the analysis of different models used during the same phase (e.g., checking the consistency of a scenario and a complete behavioral description of a component) as well as different phases (e.g., checking a bus communication schedule against the abstract communication behavior of the corresponding abstract component). This leads to a higher level of product quality, especially by supporting analysis of the system at ear-

lier stages; additionally, it also increases process efficiency by supporting earlier detection of defects.

- **Integration by Generation:** The transition between different models in the development process is supported by generating models out of each other; forward generation (e.g., the generation of test cases from a behavioral description) as well as backward generation (e.g., the generation of an abstract scenario-based description from an execution trace of the implementation level). This leads to increased process efficiency by a higher degree of automation as well as increased product quality by eliminating defects introduced by manual development steps.

With AutoFOCUS [HSE97], Quest [BLS+00], and the extended functionality of AutoFOCUS2 [SPH+02] prototypes have been and are developed demonstrating the applicability of those properties.

References

- [BLS+0] P. Braun, H.Lötzbeyer, B.Schätz, O.Slotosch. *Consistent Integration of Formal Methods*. IN: Tool and Algorithms for the Construction and Analysis of Systems (TACAS 2000). Susanne Graf, Michael Schwartzbach (eds.) Springer Verlag
- [BMJ+96] T. Bruckhaus, N. Madhavji, I. Jannsen, J. Henshaw. *The Impact of Tools on Software Productivity*. IEEE Software, 9. 1996.
- [HSE97] F. Huber, B. Schätz, G. Einert. *Consistent Graphical Specification of Distributed Systems*. In: FME '97: 4th International Symposium of Formal Methods Europe, Lecture Notes in Computer Science 1313, pp. 122 - 141, John Fitzgerald, Cliff B. Jones, Peter Lucas (ed.), Springer.
- [Jon91] C. Jones. *Applied Software Measurement*. McGraw-Hill, 1991.
- [SPH+02] B.Schätz, A. Pretschner, F.Huber., J. Phillips. *Model Based Development*. Technical Report. TUMI-0402. Technische Universität München. 2002.
- [SHH+03] B. Schätz, T. Hain, F. Houdek, W.Prenninger, M. Rappl, J. Romberg, O. Slotosch, M Strecker, A. Wisspeintner. *CASE-Tools for Embedded Systems*. TUM-I0309. Technical Report, TU München, 2003.