CASE Support and Model-Based Development

Bernhard Schätz, Jan Romberg, Oscar Slotosch, Martin Strecker

TU München, Fakultät für Informatik, Boltzmannstr.3, D-85748 Garching (Munich)
{schaetz,romberg,slotosch,streckerm}@in.tum.de

Model-based Development: State of the Art

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, ASCET-SD, AutoFOCUS, MATLAB/StateFlow, Rose RealTime, Rhapsody in MicroC, Telelogic Tau, Trice Tool). Besides providing the abovementioned support, these tools for embedded systems additionally offer support

- to describe the system using hierarchic views
- to use message/signal-based communication, including timing aspects
- to check the model for inconsistencies, mainly on the syntactic level
- to simulate the system at the level of the description .

There are, however, significant differences between the offered functionalities, generally related to the application domain the tool is targeting. They differ concerning the level of abstraction from the implementation they use. Some tools rather consequently use this abstract model; others at least partly keep the implementational view, modeling components, e.g., by object communicating by method call. Furthermore, tools generally support only either event-driven or time-driven models, focusing either on the reactive or the control part of an embedded system without integrating those application domains. Besides the underlying model, they also differ concerning the offered process support, especially concerning the consistency mechanism. Generally, all tools support syntactic consistency (e.g, well-formedness of trigger conditions). Tools do generally not support semantic consistency with few exceptions exist (e.g., a non-determinism check, checking whether a sequence diagram can be executed by state machines).[1]

Model-based Development: Picture of the Future

To increasing both the *efficiency of the development process*, and the *quality of the development product*, model-based development should exceed the following:

- **Using an implementation-level model** leads to a limited development process. E.g., defect analysis is limited to implementation level defects, excluding simple defects like interface incompatibility between processes on different nodes as those mes-

[1] The complete comparison can be found in B. Schätz, T. Hain, F. Houdek, et al. *CASE-Tools for Embedded Systems*. TUM-I0309. Tech. Report, TU München, 2003.

sages are described as a byte-block oriented bus-protocol. Furthermore, design specifications are not always related to the implementation leading to inconsistencies between design and implementation.

- **Using an OO-model for embedded software** offers only limited abstraction from the OO operational model; e.g., method calls are used to model interaction between tasks rather than more suitable message- or signal-based communication. Furthermore, domain-specific aspects (e.g., bus schedules,) are not modeled explicitly. 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 domain-specific draw-and-generate tool** allows a more detailed model including aspects like preemption or bus slots, used to generate deployable code. However, sophisticated analysis techniques are not available on the level of the model; analysis of defects introduced on the level of the model is performed manually or delayed until execution of the generated model. Thus, e.g., the tool does not ensure consistency between time-driven communication and allocated bus slots.
- **Using a loosely coupled tool chain** generally splits the development process in tool-related phases with substantial gaps in between; e.g., scenario-based descriptions of the behavior cannot be used to generate equivalent implementation test cases. The tool chain depends heavily the common model bridging the tool chain. The degree of coupling is limited by the quality of those models and their interdependence. If, e.g., a bus signal on the implementation level cannot be related back to a message between abstract components, traces cannot be expressed on that abstract level.

For optimal support, a model-based software development process needs:

- **Adequate Models:** Specific models addressing different aspects of a system (e.g., data flow, scheduling) must be available for the different phases of the development process. Besides communication and time, models must also support specific aspects of the application domain (e.g., task, schedule, ECU, bus). By supporting domain/application specific elements, stronger analysis/generation techniques are available (e.g, generating a bus schedule from the communication of a component model).
- **Abstract Models:** Models should contain only those aspects needed in the development phase they are applied for (e.g., modeling component interaction by messages rather then calls to the operating system). Abstract models reduce the possibility to produce faults (e.g., ensuring type correctness between ports rather than using byte blocks of the bus level). They also support analysis/generation at early stages (e.g., behavior completion to introduce standard behavior for undefined situations).
- **Integration by Analysis:** This includes the analysis of different models used during the same phase (e.g., checking the consistency of a scenario and a state description of a component) as well as different phases (e.g., checking a bus schedule against the abstract messages of the abstract components). This leads to a higher level of product quality, especially by supporting analysis of the system at earlier stages; additionally, it also increases process efficiency by supporting earlier detection of defects.
- **Integration by Generation:** This includes 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 efficiency by a higher degree of automation and increased quality by eliminating defects introduced by manual development steps.