Model-Based Development of Embedded Systems

Franz Huber, Jan Philipps, Oscar Slotosch

Validas AG
www.validas.de
{huber,philipps,slotosch}@validas.de

Abstract

We describe concepts and processes for model-based development of embedded control systems. Tool support for such an approach is provided by the award-winning AutoFocus/Quest tool set jointly developed by TU München and Validas AG. We outline real-time extensions for the modeling languages and show how to use time information for testing. A small case study demonstrates these extensions.

Introduction

Software development approaches that rely on modeling a system before performing the actual implementation work have a long history in computing. Among the first ones were data(base) modeling approaches using the Entity/Relationship model and similar other techniques. During further development, modeling techniques became increasingly complete, covering not only data aspects, but also structural/topological and behavioral aspects of systems. Typical representatives of such full-scale modeling approaches are structured methods, such as Structured Analysis & Design, or object-oriented methods like the UML [4].

Models created in such a modeling language can serve different purposes. They can be regarded as a concise, much more formal version of otherwise informally given system requirements. In this view, they serve as a precise guideline for the developers that perform the actual implementation work, and can furthermore be used as a basis for testing the conformance of the implementation with the requirements.

If a modeling language is rich enough to allow the creation of *complete* models (models that encompass all aspects of a system on an abstract, implementation-independent level), another purpose of such models is obvious: The created models can not only be used to precisely capture the requirements upon the system, but to describe the system in detail, reaching up to a complete description of all aspects of the system. From such a complete description, it is basically possible

(although not always feasible or desired in practice) to generate a complete system implementation automatically. An important advantage of such a model-based approach is (programming) language independence: Modeling languages are usually driven by the application domain that they are used in and provide application-oriented abstractions to describe systems (components, data entities, states, state transitions, etc.). In contrast, typical programming languages such as Ada or C are general-purpose languages, providing language elements that reflect the underlying machine model of sequential execution of statements. Using code generation techniques to create implementations, such complete models as described previously can be transformed into implementations in arbitrary programming languages.

Models are abstractions of a system and are thus particularly less "cluttered" than an implementation, for instance, in C. Therefore, it is much more promising for models than for implementations to apply validation techniques, such as—covering different levels of formality—prototyping and simulation [7], test case/test sequence generation [12], or model checking [9]. If the elements of a modeling language have been chosen carefully enough to keep the modeling language simple, yet complete, it is feasible to provide a sufficiently streamlined formal semantics that even allows the application of rigid formal validation/verification techniques [5].

Subsequently, we introduce such a simple, yet powerful modeling language—the AutoFocus modeling language & framework [8]—and outline some of the validation techniques that can be applied to AutoFocus models. The AutoFocus modeling language has been under development since 1995, specially aimed at the development of embedded systems, and shares some concepts with UML/RT.

Model-based Development Concepts

A modeling language—quite similar to a programming language—comprises a set of concepts that are used to describe systems. In case of programming languages, these concepts are typically statements, blocks, procedures, functions, and many more. For the AutoFocus modeling language and toolset, these concepts are based on the idea of a system being made up of a network of communicating components. Usually, the concepts that describe a modeling language are defined in a so-called meta-model (i.e., a model that describes how models in that modeling language can be constructed). A simplified representation of the AutoFocus meta-model is shown in Fig. 1, using the UML class diagram notation as the meta-language.

AutoFocus Modeling Concepts

The core modeling concepts of AutoFocus, i.e., the core elements in the its meta-model are as follows:

Components. They are the main building blocks for systems. Components encapsulate *data*, *internal structure*, and *behavior*. Components can communicate with their environment via well-defined interfaces. Components are concurrent: Each one of them runs sequentially; however, in a set of components, each component's run is independent of the other components' runs. Components can be hierarchically structured, i.e., consist of a set of communicating sub-components.

Data types. They define the data structures used by components. Data types are constructed from a set of basic types (such as integer or float) and a set of constructors, e.g., for record and variant types.

Data. Data elements are encapsulated by a component and provide a means to store persistent state information inside a component. Data elements can be regarded as typed state variables.

Ports. They are a component's means of communicating with its environment. Components read data on input ports and send data on output ports. Ports are named and typed, allowing only specific kinds of values to be sent/received on them.

Channels. They connect component ports. Channels are unidirectional, named, and typed, and they define the communication structure (topology) of a system.

Control States and **Transitions.** These elements define the control state space and the flow of control inside a component. Each transition connects two distinct controls states (or one control state with itself, in case of a loop transition) and carries a set of four annotations determining its firing conditions (its "enabledness"):

- *pre-conditions* and *post-conditions*, which are predicates over the data elements of the component to be fulfilled before and after the transition, respectively, and
- *input* and *output patterns*, determining which values must be available on the component's input ports to fire the transition and which values are then written to the output ports.

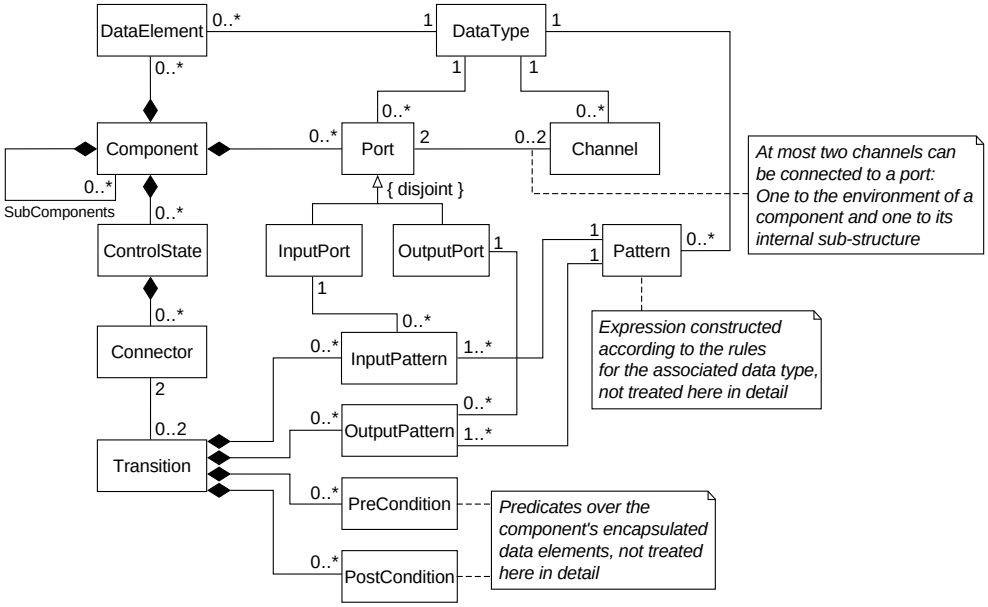

Fig. 1. Basic Modeling Concepts of AUTOFOCUS: The Meta-Model

These concepts are sufficient to describe a large class of systems. Developers create the model of an actual system using these concepts; technically speaking (e.g., with a modeling *tool* for this language in mind), an actual system model is an *instance* of this meta-model. The complete meta-model, together with a set of additional conditions relating to consistency and completeness of models, describes the set of all possible, well-formed models that can be created.

Views and Description Techniques

Developers do not create and manipulate models as a whole, but by picking only specific parts of them, which are of interest during particular development activities. These parts, usually closely

related with each other, make up the *views* of the system. For instance, the structural view in AUTOFOCUS considers only elements from the meta-model describing the interface of components and their interconnection.

To manipulate elements of these views we must represent them visually. In AUTOFOCUS we use mainly graphical notations for that purpose; these notations are introduced in more detail by our application example. The notations do not represent self-contained documents; instead they are a mere visualization of a clipping from the complete model. Fig. 2 shows an example for this relationship between a set of related elements from the meta-model (inside the shaded area) and their visual, diagrammatic representation. In this example, structural aspects of the model are covered, and the notation used to visually represent them is called *System Structure Diagrams* (SSDs for short).

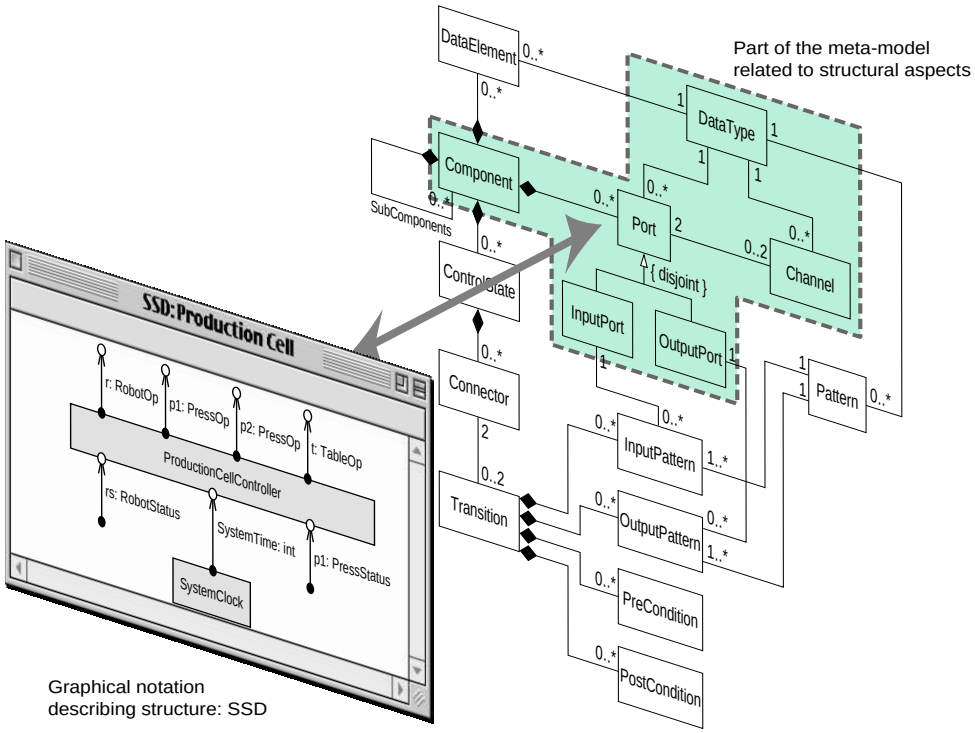

Fig. 2. Structural Parts of the Meta-Model and the Notation representing them

Model-based Development Process

One of the main difficulties in software engineering is that the requirements of the customer are prone to change while software is being developed. In standard waterfall development process models, which are still regularly used in industry, requirements analysis and testing are located at opposite ends of the development process. Evolutionary development processes, on the other hand, try to alleviate this problem by building the software system *incrementally*. Requirements are not fixed in an early development phase, but instead converge during several incremental cycles with customer interaction after completion of each increment.

In this section, we first give a short overview over incremental development processes, and then describe how a process based on executable models is supported by modern CASE tools, such as AutoFocus. Finally, we explain the step from executable models to final code.

Incremental Development

Boehm's spiral model [3] is the most famous incremental process model, although it is more a meta-model of a process than a proper development process model. More helpful for real software development are the Cleanroom Reference Model (CRM) [11], and so-called agile approaches, most notably Extreme Programming (XP) [2], which is based on classical object-oriented programming languages.

We believe that modeling languages fit the demands of an incremental process better than programming languages: Their higher level of abstraction leads to higher productivity of the developers; their suggestive notations ease interaction with the customer and other developers. Nevertheless, models are executable, which results in immediate feedback for the designer and the customer.

The idea of incremental model-based development amounts to specifying the model of the system as precisely as possible, so that the model is always executable. In order to handle the complexity of the system, in a first step only a small part of the core functionality of the system is described. The specification is then (together with the customer) validated and verified by simulation, inspection and reviews, and by formal verification and analysis techniques. Later steps refine this model: More components are added to the model in order to add functionality; the behavioral specifications of the components are elaborated to handle exceptional cases; additional inputs and outputs are added, for instance to ease maintenance of the final product.

Besides the modeling activities themselves, the process consists of the following activities:

- **Simulation**: Model executability is the basis of the main validation technique employed in our incremental process [7]. Together with the customer, exemplary system runs are produced that demonstrate the model essentially operates according to the customer's requirements.
  Simulation is not restricted to interactive step-by-step executions. Using advanced symbolic execution techniques based on constraint-logic-programming, it is possible to automatically derive simulation runs from abstract test case specifications; a test case specification typically demands that the model is brought into a certain state (functional tests) or that every transition is executed at least once (structural tests).
- **Analysis**: While simulation is helpful to determine that the system indeed fulfils its requirements, there are some questions related to quality assurance that cannot be answered by simulation alone, since simulation gives answers only about *single* system runs, not about *all possible* system runs (mathematically, simulation shows existential properties, not universal ones). Some typical questions are whether the model is *deterministic* (i.e., for each input from the system environment there is *at most* one possible output specified) and *complete* (i.e., for each input from the system environment there is *at least* one possible output specified). The AutoFocus toolset includes analysis tools that help to answer such questions. It also includes verification tools such as model checkers [9], which are used for mathematical proofs of critical system properties. Since such proofs are very expensive (in terms of time, effort and required expertise of the tool user), the use of verification tools must be carefully judged against the economic risk of system malfunctions.
- **Refactoring**: An obvious problem with any incremental system development process is that the resulting system specification may be cluttered and hard to understand, as its

structure is determined partly by the order in which the increments occurred. Extreme programming makes use of elaborated *refactoring* [6] patterns to clean up the system after each increment so that it is both easier to understand and more amenable to further increments. Similar techniques can be used for executable models; however, this is still an active area of research.

To summarize, we advocate a development process that consists of several iterations where at the end of each loop, instead of hand-written code an executable system model is presented to the customer. This approach is similar to Extreme Programming, but focuses on a more abstract modeling of the system rather than its implementation. In contrast with Extreme Programming, however, production of the final code is deferred until the end of the development process.

From Models to Products

Once the model is considered to be sufficiently correct and detailed, it is used as the basis for the production of the target code. For the target code, too, quality assurance must be performed. As the resulting code is likely not amenable to automatic analysis, the core activity here is testing (see, e.g., [10]). Essentially, there are two approaches:

- The target code is produced by hand. This is a typical situation for customer/supplier relationships, where the model serves as the software specification of the final product. In this case, the code produced by the supplier must be tested to ensure its conformity with the model. It is possible to automatically derive test sequences for the implementation from simulation runs, in particular from the runs produced by symbolic execution as mentioned above.
- The target code is produced by an automatic code generator. While in principle it is possible to mathematically verify code generators, in practice there is still some risk (albeit a very small one) that the code generator produces incorrect code. Thus, even for automatically generated code it is prudent to test the code. For avionics systems, rigorous testing is even required: Standards such as DO-178B require, among other points, tests with clear code coverage criteria (MC/DC); it is not sufficient to have coverage only on the models.

In both cases, however, additional tests must be performed to ensure that the model is not based on incorrect assumptions about the interaction with the environment, which could lead to timing problems and race conditions. Such tests can be performed by Hardware-in-the-loop approaches.

Example: A Digital Watch

As an example of the description techniques of AutoFocus, this section presents parts of a model for a digital watch. The watch has three buttons (T1, T2, T3) that are used to change the display mode (date, time, stopwatch) and to set the current time and date after a battery change. Fig. 3 shows the top-level structure diagram of the watch model.

The data types of the channels are defined using Data Type Definitions (DTDs) as follows:

```
data Signal = Present;
data Segments = Date(Int,Int,Int,Int,Int,Int)
              | Time(Int,Int,Int,Int,Int,Int)
              | Stop(Int,Int,Int,Int,Int,Int);
```

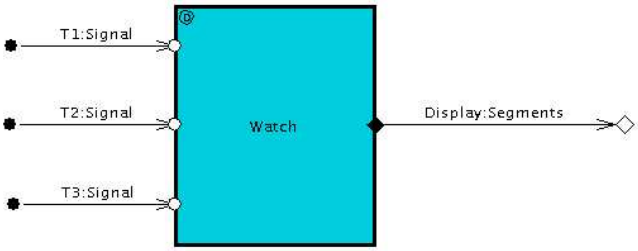

Fig. 3. Top-Level View of the Watch

Deeper in the modeling hierarchy, the watch component contains a time component that computes the time of day from internally derived signals (**hs**, **zs)** that hold the time of day in 1/100s and 1/10s; they are derived from an internal counter. There are other components to keep track of the current date and to model the stopwatch function.

The time component (see Fig. 4) is quite complex, because it is also responsible for the adjustment of the time by the user. It has separate components to compute the segment digits for the display (**Sec1, Sec10, Min1, Min10** and **H12**); depending on the current display mode they either display the relevant part of the current time, or the current time setting when the watch owner changes the time.

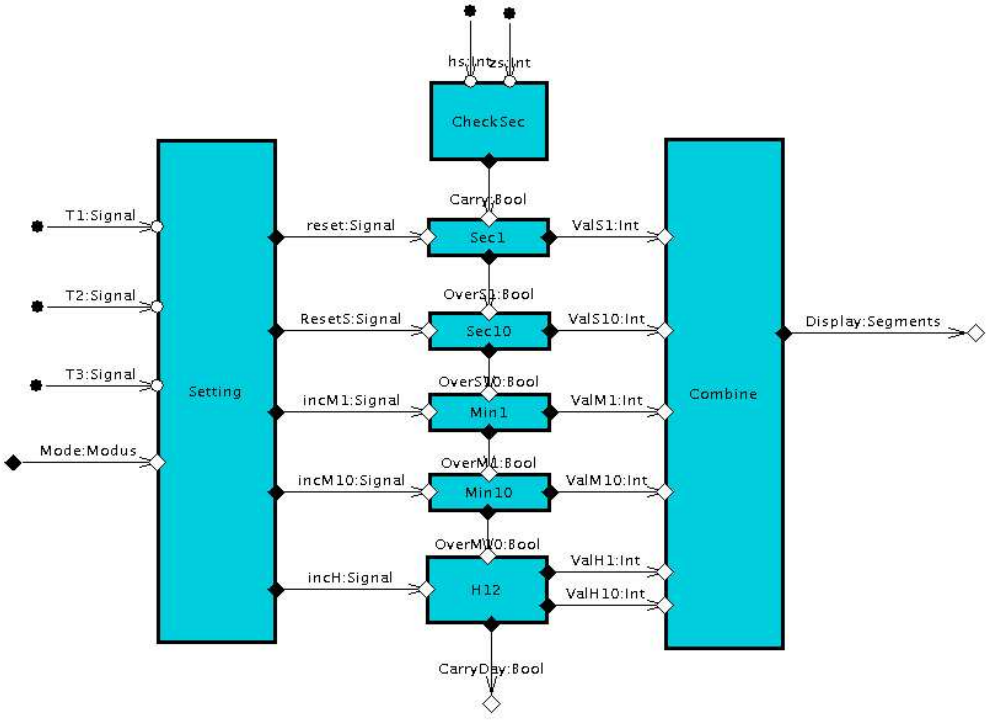

Fig. 4. Component Network inside the Time Component of the Watch

Note that in the diagram there are two kinds of channel connectors: *Delayed* channels (marked with a circle) and *immediate* channels (marked with a diamond). Messages sent through an immediate channel are visible to the receiver in the same reaction; messages sent through a delayed channel are visible in the following reaction. There are some subtle methodical issues involved in

their use. Generally, immediate channels should be used for the data flow within an embedded controller. Without immediate channels it would not be possible to switch from 23:59:59 to 00:00:00 within one model reaction, because the overflow-values would be present only in the next time step. On the other hand, for mathematical reasons every communication cycle in the system must contain at least one delayed channel.

Fig. 5 shows the behavior of the component **Setting** as an example for state transition diagram. It describes the way to set the time using the signals **T1, T2, T3** and the current mode that is computed by another component of the model (the mode signal is also computed from the three buttons, but it is handled separately for reasons of modularity). For example, incrementing the minutes display is modeled with the transition **T2?P; T3?; IncM1!Present** which connects the state **Minute1** to itself. The semantics of this transition is as follows: When button **T3** is not pressed, but button **T2** is, then send the signal **Present** to the minute segment component.

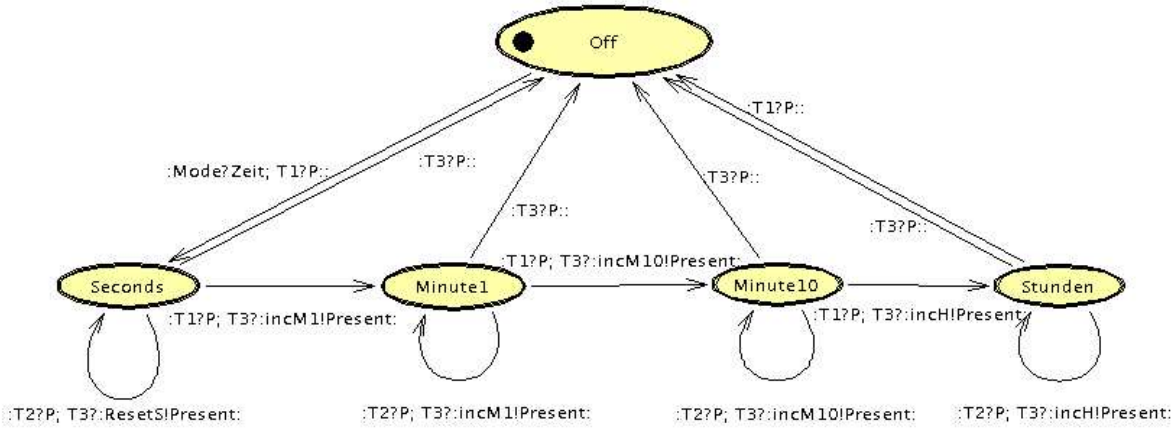

Fig. 5. State Transition Diagram of the Setting Component

In addition to input and output statements transitions can be annotated with preconditions (which are predicates over local variables and values read from input channels) and actions (which are assignments to local variables). Together with the DTD specifications, this allow models that are completely independent from the target programming language. This is important for reusing the models for other targets (for example with a different processor and a different instruction set).

While the model itself is not very ambitious, its implementation is, since the resources of the watch are severely limited (4bit processor, 2KB of memory). The model is indeed the basis of a watch implementation with an industrial partner, but we did not yet implement a code generator for the assembly language used in the project. Given the rigorous semantics of AutoFocus, which is very close to the description techniques, and the limited size of the watch model, it is still manageable to translate the model into assembly language by hand.

The code that results from the watch model is not used stand-alone; it is linked with a number of arithmetic and I/O libraries.

Extensions for Real-Time

With the description techniques presented so far, models describe the functionality of the system under development in terms of input/output reactions. For the final implementation, in addition to

the functionality, timing requirements must be considered. They relate system executions with the physical time of the system's environment. Typical timing requirements are stated as *separation* requirements (two inputs or outputs must be separated by at *least* a certain duration), or as *proximity* requirements (two inputs or outputs must be separated by at *most* a certain duration). Timing requirements fulfill two purposes: They impose demands on the maximal execution time for an input/output reaction of the system (and thus on the performance of the target hardware), and they state assumptions on the duration of activities of the controlled hardware.

Timing requirements can be concisely specified by annotated Message Sequence Charts. Fig. 6 shows a simple timing requirements for the watch: It states that the watch, when switched to date mode by pressing button T3, displays the date for three seconds (rather, any number of display messages are sent to the hardware until exactly 3 seconds have passed) and then returns to time mode. This requirement is a combination of proximity and separation requirement. Further (quite trivial) requirements specify that a second (rather, 1/100s) of model time corresponds to a second of real time.

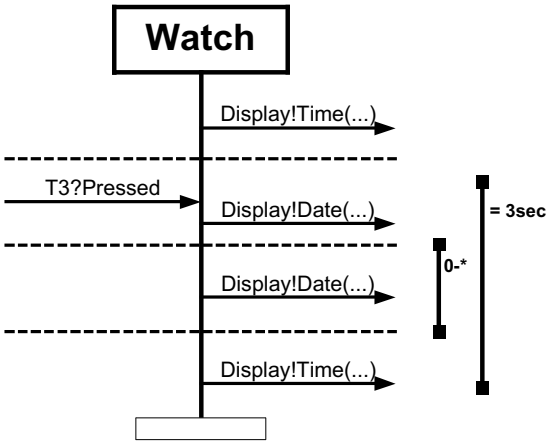

Fig. 6. Timing Requirements in a Message Sequence Chart (MSC)

Obviously, the relationship between model time and real time must be verified. Currently, this is done by hand-written test cases, which measure controller execution time and environment response times. A more automatic scheme will translate time-annotated MSCs like the one above into special time observer components. These components run in parallel to a standard hardware-in-the-loop setup (see Fig. 7) and give verdicts on the satisfaction of timing requirements for not only special timing test cases, but for all previously derived functional test cases as well.

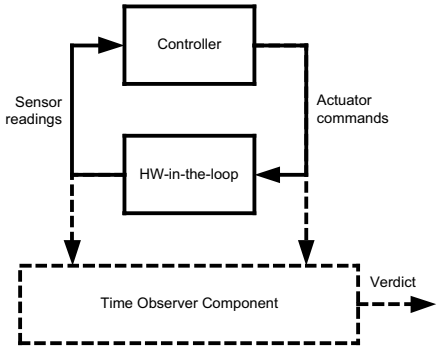

Fig. 7. Time Observer Component in Parallel with Hardware-in-the-Loop

Conclusion

This paper touches only some of the issues of model-based development. While the classical areas of CASE tools (description techniques and code generators) are quite stable in our toolset, the validation and verification tools are still undergoing development and field studies with our partners and customers. In particular, the promising field of model-based test case generation is making rapid progress. In order to cover not only the later phases of system design, but also requirements analysis, the AutoFocus toolset has been connected to the requirements management tool DOORS [1].

Acknowledgment. Peter Braun, Heiko Lötzbeyer, Alexander Pretschner and Dr. Bernhard Schätz, our colleagues from TU München, have contributed greatly to our understanding of model-based development and to the AutoFocus toolset itself.

References

1. B. Bajraktari: Modellbasiertes Requirements Tracing. Master thesis, TU München, 2001.
2. K. Beck: Extreme Programming Explained: Embrace Change. Addison-Wesley, 1999.
3. B.W. Boehm: A spiral model for software development and enhancement. Software Engineering Notes, 11(4), 1994.
4. G. Booch, I. Jacobson, J. Rumbaugh: The Unified Modeling Language User Guide. Addison-Wesley, 1998.
5. M. Broy, O. Slotosch: Enriching the Software Development Process by Formal Methods, *Proceedings of FM-Trends 98*, LNCS 1641.
6. M. Fowler: Refactoring. Improving the Design of Existing Code. Addison-Wesley, 1999.
7. F. Huber, S. Molterer, A. Rausch, B. Schätz, M. Sihling, O. Slotosch: Tool supported Specification and Simulation of Distributed Systems, *Proceedings of International Symposium on Software Engineering for Parallel and Distributed Systems,* 1998.
8. F. Huber, B. Schätz: Integrated Development of Embedded Systems with AutoFOCUS. Technical Report TUM-I0107, Fakultät für Informatik, TU München, 2001.
9. J. Philipps, O. Slotosch: The Quest for Correct Systems: Model Checking of Diagrams and Datatypes, P*roceedings of Asia Pacific Software Engineering Conference 1999*, 449-458.
10. A. Pretschner, O. Slotosch, H. Lötzbeyer, E. Aiglstorfer, S. Kriebel: Model Based Testing for Real: The Inhouse Card Case Study in Proc. 6th Intl. Workshop on Formal Methods for Industrial Critical Systems (FMICS01), Paris, July 2001.
11. S. Prowell, C. Trammell, R. Linger, J. Poore: Cleanroom Software Engineering. Addison-Wesley, 1999.
12. G. Wimmel, A. Pretschner, O. Slotosch: Specification Based Test Sequence Generation with Propositional Logic, *Journal on Software Testing Verification and Reliability* (to appear).