Enriching the Software Development Process by Formal Methods*

Manfred Broy, Oscar Slotosch

Institut für Informatik, Technische Universität München
80290 München, Germany

Abstract. We describe a software development process designed for an integration and usage of formal methods into practical software process models in a scalable way. Our process model is an extension of the V-model, and allows the specification of critical components and the verification of crucial development steps. For different development stages we suggest user-oriented description techniques, based on a common formal semantic. Furthermore we outline methods for the verification of critical development steps. We illustrate our process by developing a small example with some critical aspects.

1 Introduction

The development of software systems is a difficult and error prone task. This is certainly true if systems get very large and complex. However, this may even be true in cases where small to medium size programs have to be developed that are based on complex algorithms, data structures, or patterns of interaction.

Today software development in practice is almost always done in a non-scientific manner [Hoa96] based on pragmatics and heuristics. The development process is structured into phases like:

- analysis and requirements engineering,
- specification,
- design,
- implementation and testing.

In most cases, large parts of the development work are done informally and the correctness of the developed systems relies mainly on the intuition and experience of the developers received by extensive inspections, testing, and prototyping. The debate is still going on whether there is a cost effective alternative to this heuristic approach to software development including and applying so-called formal methods.

In the early days of computing science there was not even a theoretical alternative. The scientific foundation of programming computer systems were not

* This work was supported by the German Information Security Agency (BSI) within the project Quest.

understood. Programming languages were seen as pragmatic, operational notations to control the machine, used on the basis of an intuitive understanding. For many of the programming concepts a theoretical scientific foundation was not available.

Today the theoretical situation has changed. Over the last 30 years the scientific community of computer scientists has developed a solid foundation of software and systems engineering [BJ95,Bro95]. Today there is nearly no programming construct or development concept for which a scientific basis is not available. Of course, still a lot of notation is used in practice for which such a scientific foundation is not worked out explicitly. But this is not due to the lack of theory but rather it has not been done since those people suggesting the notation to a large extend have not yet recognized the virtue of a scientific foundation. For an improvement of the formal basis of software development, it is therefore highly important to have a clear idea what formal methods are good for and how they are combined with informal ones in a way to get the best benefit out of this combination.

Our paper is organized as follows. After a clarification of the term "formal methods" (Section 2), we describe the role of formal methods in the development process and how formal techniques can be combined pragmatically within formal methods. In Section 3 we start with a short presentation of the V-model, and show how formal methods can be integrated into the development process in a scalable way by using them only for the critical parts and for important development steps. In the rest of this paper we illustrate the development process by means of a small example which is presented in Section 4. Section 5 describes different views of the system during the development process, especially in the requirement, design and implementation phases. Furthermore we present graphical description techniques that can be used in a co-development with formal and conventional methods since all description techniques are based on the same semantic model. In Section 6 we present methods that support the engineer in proving the correctness of development steps between different views, and we analyze tool support for different refinement situations.

2 What is a Formal Method

In spite of the enormous amount of work spent on formal methods the notion of a "formal method" is not always made sufficiently precise [Bro96]. We therefore clarify this to begin with. First of all, let us define what we call a method in computing science.

A *method* consists of description concepts, rules for constructing and relating descriptions, and development strategies explaining how and when to apply the method in a goal directed manner. We therefore refer in the following to three ingredients of a method:

D: description concepts,
R: rules, and
S: strategies.

For each of these ingredients we can ask to which extend they are formal in order to evaluate the degree of formality of the method.

Looking at description techniques D we identify the following aspects:

- syntax,
- semantics, and
- proof rules.

A description technique is *fully formal* if it has a *formal syntax* (including both context free syntax and context correctness), a *formal semantics* (described in a precise mathematical way), and a *logical theory* that supports formal proofs about the descriptions and their properties.

A description technique is called *semiformal* if it has a formal syntax but not a formal reference semantics nor proof rules.

The rules R of a method describe the rules for constructing and relating descriptions correctly. For a formal method R consists of proof and transformation rules for the refinement between descriptions, and of (consistency) rules describing the correct construction of descriptions. For a fully formal method we require that the proof rules are given as a formal calculus or embedded by proof obligations into a common logical theory such as first order predicate logic or higher order logic. The same must hold for the transformation and consistency rules.

Finally, a formal method should formalize its development strategies S. Of course, we cannot expect, in general, that the strategies are formalized to a point where the strategies are given by algorithms, such that they can be carried out completely mathematically. In general, the developer guides and controls the development process. We require, however, that the phases and steps are characterized in a formal way, such that it is made precise when a development process is in fact an instance of the overall development strategy.

3 V-Model

The V-model [BD95] is an ISO standard for structuring the software development process. It is supported by the German Ministry of Defense. It contains many detailed steps, guiding the developer through the software engineering process. The name V-model origins from the main scheme of the model, which is depicted as a V (see Figure 1). The left hand side of the V represents the waterfall model of software engineering from requirements down to the program code. The right hand side describes the quality control activities: testing of the components, integrating them, and testing the whole system. The horizontal (dotted) lines express that the requirements of the system have to match the result of the integration test, the integration of the parts has to compose the system according to the structure fixed in the decomposition during the design.

The model is independent from a specific programming or modelling language, and therefore it is used in many developments projects. It supports the development of components, since the activity of testing the components follows

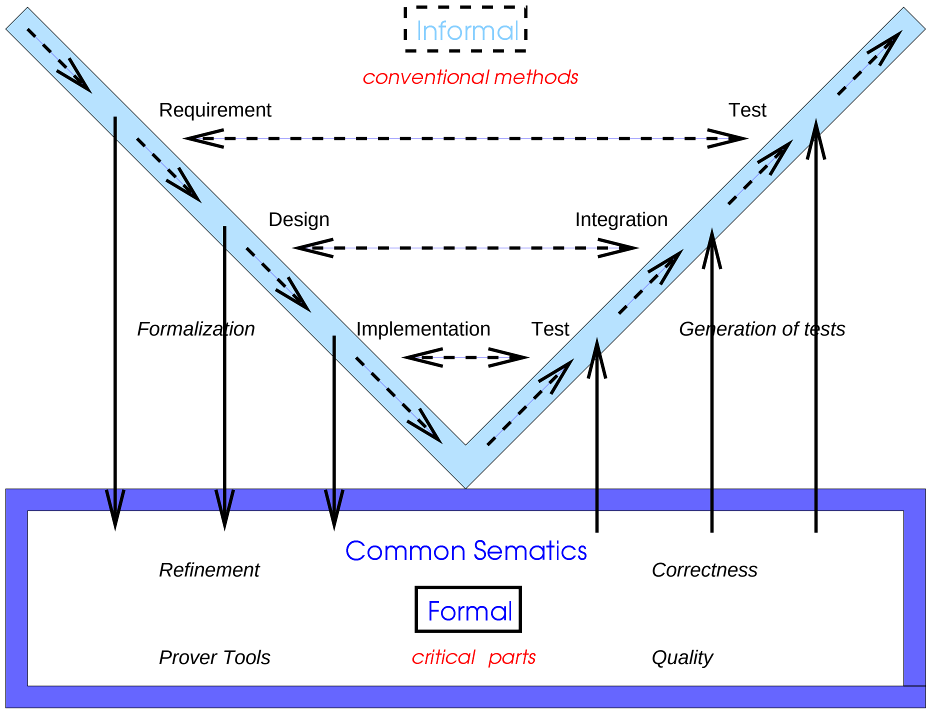

Fig. 1. process model with formal methods

directly after the coding, and errors can be corrected immediately. However, errors made during the decomposition can only be detected during the integration of the system. And errors in the requirements are even more expensive, since they are discovered at the last point, where the whole system is tested.

Formal methods promise that extensive testing is not required if these methods are applied consequently. However, since a complete formal development is not feasible for large projects, the hope for a better process model decreased. We advocate a scalable use of formal methods, by integrating formal methods into the V-model in order to reduce the number of expensive errors and of the test overhead.

Our core concept is to work with user-oriented description techniques, basing on a common formal semantic, to specify critical parts of the system. This basis allows us to ensure the correctness of important development steps, during all phases of the software development process.

There are two degrees in which the conventional development process can be enhanced by formal methods:

– Using formal description techniques in the development process for the modeling of the system views provides these views with a clear denotation and

reduces the number of misunderstandings that occur due to ambiguities of specifications [GSB98]. To facilitate the development process, the applied formal description techniques should be similar to those techniques established in practical methods.
- Using formal refinement to ensure the correctness of critical development steps is the fully formal way for proving the correctness [BJ95,Hin98]. There are two ways in doing this: the *deductive method* allows the developer to do any step, provided that its correctness can be deduced (after the step). The *transformational method* restricts the developer a priori to the set of applicable transformation rules. The first, more flexible, method requires to model the abstract and the refined view and generates proof obligations, whereas the transformational approach requires to integrate the result of the transformations into the development process. In case that the semantics are given by a translation into a calculus, there is a retranslation required to continue with the development on the conventional side. By allowing the developer to prove the correctness of new transformation rules the transformational method can be seen as a special case of the deductive method.

With these different levels of formality, the use of formal methods in the development can be scaled according to the requirements of the project.

We recommend graphical description techniques with formal semantics (see Section 5.5, [Bro95]). The formal semantics allow us to systematically generate test cases from the specifications (Section 6.4, [Sad98,Par95]). This is a further reason for using our formal description techniques in development projects.

4 A small Example: Traffic Lights

We develop the software for a traffic light system that controls a pedestrian crossing. Pedestrians operate two buttons (one at each side of the street) to request the green light. In addition, there are acknowledge lights on both sides of the street to indicate the pedestrians that their request has been excepted by the system.

The requirements for the software to make the system behave like a usual crossing (e.g. both pedestrian and both car lights show consistent signals) are assumed to be realized by the design and the implementation of the system. As part of those functional requirements there are some critical aspects of the system, which have to be guaranteed by the developer:

NO-CRASH: It must be excluded that pedestrians and cars have green lights at the same time.

NO-BLOCK: It must be excluded to block the traffic, i.e. neither pedestrians, nor cars have to wait forever.

SEPARATE: After the red phase for cars there is a yellow signal before the traffic lights are switched to green. After the green phase there is again a yellow light.

In fact, the mentioned requirements are of different nature. The first and the last are are safety conditions, while the second one is a lifeness condition.

5 Formal Description Techniques for System Views

In this section we introduce formal description techniques that occur at different stages during the development process and focus on different views of the system, according to the requirements of the software development process at this stage.

We present graphical description techniques (see [HMR+98]) with common formal semantics for the modelling of the different views. The common formal semantics allows us to define the relation between the different views and the different development stages to support the formal development (see Section 6). The techniques are summarized in Section 5.5 to illustrate the different techniques and to show how the degree of formality can be scaled we use the example of Section 4.

5.1 Requirements Specification

The first views of a system within the development process occur in the requirements specification. The requirements specification covers the following aspects:

- Boundaries of the system to its environment.
- Interactions between the system and its environment (communication channels and messages). The interactions may be incomplete descriptions of the behaviour of the system.
- Critical parts of the system (together with their interactions).

In our example we start with a data model for the messages which we define in a textual notation:

```
data PedLight = PRed | PGreen
data CarLight = CRed | CYellow | CGreen
data Switch = On | Off
data Signal = Present
```

Note that the channels may be empty, therefore we need no signal `Absent`. Figure 2 shows the structure of the system, which contains (non-critical) components to merge (ButMerge) and to split (AckSPlit, CLSPlit, PLSplit) the messages. The only critical component is the controller, which is developed more formally.

The critical requirements, which the controller has to fulfil in any case, are formalized using temporal logic (with the until-operator $\cup$ and the next-operator $\circ$).

NO-CRASH: $\Box\ \neg(\mathrm{CL}{=}\mathrm{CGreen} \wedge \mathrm{PL}{=}\mathrm{PGreen})$
NO-BLOCK: $\Box\ \Diamond\ \mathrm{CL}{=}\mathrm{CGreen} \wedge \Box\ (\mathrm{But}{=}\mathrm{Present} \Rightarrow \Diamond\ \mathrm{PL}{=}\mathrm{PGreen})$
SEPARATE: $((\mathrm{CL}{=}\mathrm{CGreen} \cup (\mathrm{CL}{=}\mathrm{CYellow} \wedge \circ\mathrm{CL}{=}\mathrm{CRed})) \cup \mathrm{CL}{=}\mathrm{CRed})$ $\cup\ \mathrm{CL}{=}\mathrm{CYellow}$

Of course, there are many more requirements that are less safety critical such as that the pedestrian light only gets green if the button is present.

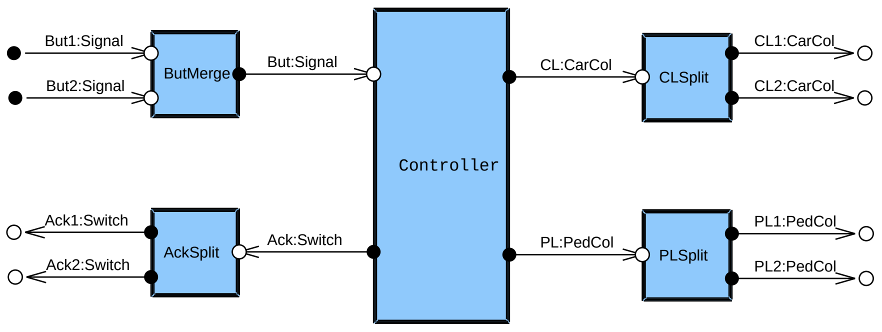

Fig. 2. Structure of Traffic Light System

Note that in **NO-BLOCK** $\diamond$ PL=PGreen does not always hold. If no pedestrian arrives, the light will never become green. The specification of temporal logic formulas, especially using the until-operator $\cup$ is not an easy task. Therefore more readable notations are useful. Interaction description diagrams are an intuitive way to specify some temporal formulas. However, they are not as expressive as the temporal logic (see below).

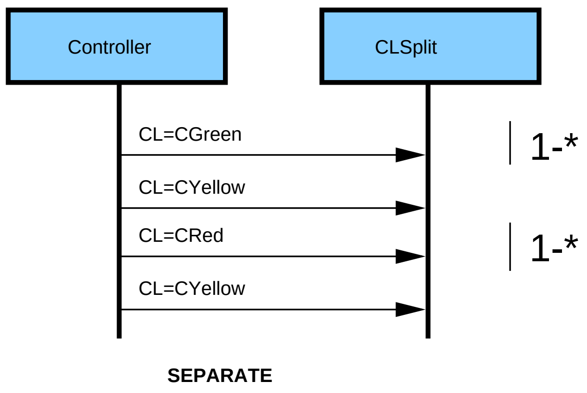

Fig. 3. Visualization of Separation Property

The critical aspect **SEPARATE** has a representation as an interaction diagram. In Figure 3 the (strong) until-operator $\cup$ is visualized by the (1 - *) modifier in the interaction diagram. Note that we omitted the outputs of the component CLSplit in the diagram, because CLSplit is not a critical component, and we therefore do not formally specify its behaviour[1]. Using interaction description diagrams like EETs (extended event traces [BHKS97]) or MSCs [IT93b] in this way is only possible if there exists a formal semantics (see [BHKS97,Bro98,Hau97] for semantics of EETs and MSCs).

The properties **NO-CRASH**, and **NO-BLOCK** cannot be represented so easily by EETs or MSCs. It is a current research topic to extend expressiveness of graphical description techniques towards more complex properties. One straightforward extension of EETs is the visualization of simultaneous actions, as developed within the project Quest. It uses time ticks to group messages together that occur within the same time interval. In our example we specify the crash-situation in Figure 4. Note that this property must not occur in our system.

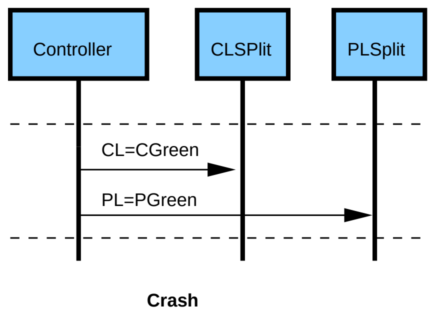

Fig. 4. Visualization of Crash Property

Deciding during the specification of requirements what a critical component is, the user can be supported by searching all components of the system that produce messages mentioned in the specification of critical properties. In our example this are the components ButMerge and Controller. We decided not to regard ButMerge as critical, since its output is used only as a premise within the property **NO-BLOCK**.

In the case of a more abstract security model it might be necessary to map the general properties to the concrete components. In our example the property that no car hits a pedestrian has been mapped informally to the controller property **NO-CRASH**. Ensuring adequacy of this mapping would require to build a model of the application domain, including assumptions to the environment, like every car stops at a red light, or pedestrians leave the crossing no later than 5 seconds after they have entered it. In our small example we do not model the environment.

[1] It is also possible to omit the component CLSplit from the interaction diagram completely, but we want to show the interaction within interaction diagrams.

5.2 Design Specification

The second class of views of developed systems occur in the design phase. They capture the structure of the system (together with all channels and data types) and a concrete description of the behaviour of the components. For the critical components we require that there is a formal design specification in order to verify that the requirements are fulfilled (see Section 6.2). To ensure quality of non-critical parts a coarse formal specification of the design of the behaviour can help to develop test cases (see Section 6.4).

We use system structure diagrams (SSDs) to specify the design structure (as in the requirements specification) of the system, and state transition diagrams (STDs) to model the behaviour of components, since they have the same mathematical basis and therefore semantically fit the techniques of the requirements specification.

In our example we refine the structure of the controller into a core controller and a timer, which is used for timing the behaviour of the system. This structure, and the communication channels are depicted in Figure 5. The timer component

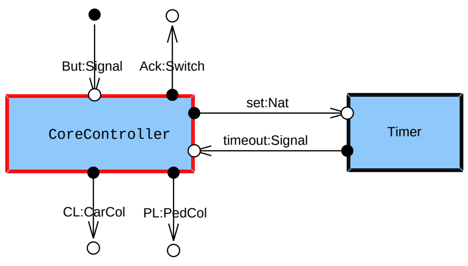

Fig. 5. Structure of Controller

uses the type of natural numbers, which is specified with the some functions working on them in the following data type specification:

```
data Nat = Zero | Succ(Nat);
fun pred(Succ(n)) = n;
```

The behaviour of the timer is specified by the STD in Figure 6, using tuples of preconditions, input, and output patterns and actions as transitions. The specification of the behaviour of the core controller is shown in the STD of Figure 7. It shows the two main states of the system: The state Wait, where pedestrians have to wait, and the state Walk where they may start walking. The transitions between the states show how the pedestrians lights are switched. The behaviour in Figure 7 is not completely designed, because it does not specify how the controller reacts on input, and how it switches the cars light. These details are modelled

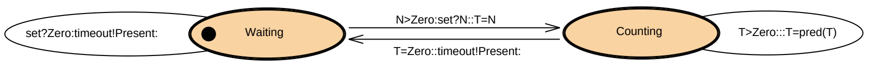

Fig. 6. Behaviour of Timer

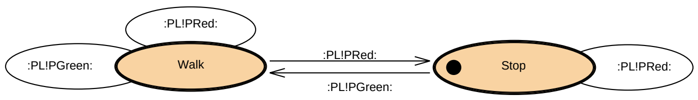

Fig. 7. Behaviour of CoreController

within the refined (sub-)STDs for the states Walk (Figure 8) and Wait (Figure 9). The introduced hierarchy is only syntactic, i.e the transitions (segments) are connected, such that the control in a hierarchic automaton is exactly in one atomic state. All transitions (segments) between different levels of abstractions are executed within one step. For example the transition from the atomic state Yellow to the atomic state Red leaves the hierarchic state Wait and enters the hierarchic state Walk and generates the output: CL!CRed,Ack!Off,PL!PGreen,set!ten

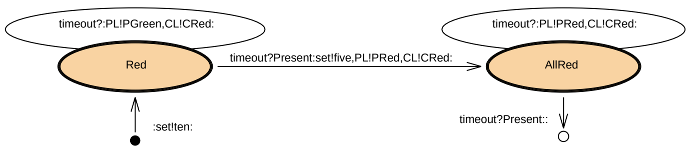
timeout?:PL!PGreen,CL!CRed:
Red
timeout?Present:set!five,PL!PRed,CL!CRed:
timeout?:PL!PRed,CL!CRed:
AllRed
:set!ten:
timeout?Present::

Fig. 8. Behaviour of Walk

Hierarchy is a very useful feature in formal notations. It allows us to specify the critical components at the right level of abstraction. Furthermore hierarchic formalisms support the development of larger systems. This is extremely important if graphical notations are used, since complex diagrams are hard to understand.

5.3 Implementation Specification

The third class of views on systems occur in the implementation specification, and describe the program code. They cannot be clearly separated from the views during the design phase. The main difference is that a design specification does

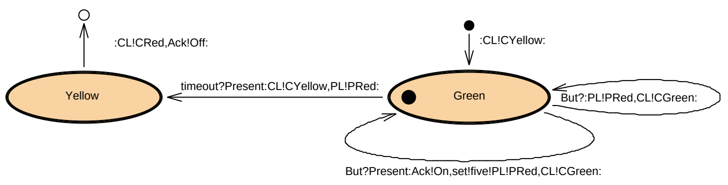
:CL!CRed,Ack!Off:
:CL!CYellow:
Yellow
timeout?Present:CL!CYellow,PL!PRed:
Green
But?:PL!PRed,CL!CGreen:
But?Present:Ack!On,set!five!PL!PRed,CL!CGreen:

Fig. 9. Behaviour of Stop

not need to be deterministic, but an implementation and its specification should be deterministic; otherwise the test results may not be repeatable.

The program code can be seen as the most detailed design specification. There are at least two different possibilities to formally specify the implementation: one is to use a programming language with formal semantics, the other way is to use the same formal description techniques as in the design specifications and to generate code out of them. Many description techniques support code generation (Statecharts [Har87], SDL [IT93a], and STDs [HS97]).

Since a common semantics for all views is required, we recommend again to use STDs for the specification of the code. In our example the detailed design specification of the core controller (Figures 7, 8, and 9) is deterministic, and we can generate code out of it. The component Timer was classified to be non-critical for the following reasons:

- It has an infinite state space (it contains a variable `T:Nat`, to store the remaining time), and therefore it cannot be model checked completely. Treating the timer as critical would require interactive theorem proving (see Section 6), which in our case is not adequate.
- Treating it as non-critical allows us to implement it manually, for example using a realtime-clock that ensures that the distance between two ticks is no less that 100 ms[2].

We restrict our implementation specifications to deterministic programs. For non-critical parts of the system any (deterministic) programming language can be chosen, the only important point is that the interfaces fit to the formal interface specification.

5.4 Test Specification

Our last class of views in the development process describe test cases. Testing components is only necessary if they are developed informally. Specifying test cases is useful for documentation purpose, and a specified test case can be reused if the implementation has changed. In Section 6.4, we show how test cases can be generated from specifications.

[2] The upper bound depends on the cycle time of our system and cannot be ensured formally within our time-free model.

As a representation of a test case we choose interaction diagrams (with time ticks), as they were presented in Section 5.1, but we do not require the full power of them, since for testing one component or one system an interaction diagram with one axis suffices. Furthermore we require to have concrete input values (no variables) on the input messages, to test the component.

5.5 Description Techniques

The description techniques used during the development process are summarized in the following table:

name	purpose	phases	hierarchy	semantics
DTD	data types	all phases	inclusion of DTDs	algebras
SSD	structure	all phases	SSDs for components	streams & functions
STD	behaviour	design, implement.	STDs for substates	IO-automata
EET	Interaction	requirement, test	EETs in boxes	predicates

Note that we did not use the inclusion of data types, nor the grouping of EETs into boxes in our example. However, since hierarchic description techniques support different levels of abstraction it is important that all description techniques are hierarchical.

AutoFocus [HMR+98] is a tool that supports the specification, with all these description techniques and has features like consistency checks and simulation. The common semantical basis of the description techniques is described in [BDD+93,Bro95].

6 Development

In the previous section we presented graphical support for the specification of the different views, which occur at different stages within the development process. Furthermore we integrated formal methods into conventional software engineering in a way that only the critical parts have to be formally modelled. The used description techniques have a formal semantics. In this section we will exploit the fact that this semantics is uniform, i.e. all description techniques have a denotation within the same semantic model. This allows us to define relations between different description techniques. The fact that these relations can be formally checked provides us with a powerful tool to develop correct software.

6.1 Checking Requirements against Design

In this section we present methods that allow us to formally verify whether a design specification meets its requirements specification. We discuss different specifications and present different techniques for tool supported correctness proofs.

The most simple case is a deterministic design and simple temporal formulas, which can be represented by EETs. This situation corresponds to a formal test. To "verify" this it suffices to feed the input values according to the specification of the test case (EET) into a simulation of the component and to compare the outputs. If the outputs fit together the test is successful. One example is the requirement **SEPARATE**, specified in the EET of Figure 3. Simulating the design specification "proves" that this requirement is fulfilled by the specification.

A slightly more difficult case is that of non-deterministic automata and non-deterministic (but finite) choices in the interaction description diagrams. It requires backtracking if the output of the component does not fit to its requirements. Backtracking checks all alternatives. If they are finite, backtracking can decide whether the requirements hold. Since the programming language Prolog supports backtracking, it seems appropriate for this form of verification[3]. If the alternatives within a program are not finite, for example due to an input value with infinite range Prolog might find a solution. But it might also not terminate, if no solution exists.

If the requirements are specified with temporal logic, especially using a negation (like in **NO-CRASH**), simulation has to inspect all cases to ensure that the requirement is valid. In this case model checkers are more efficient than Prolog is. This case is standard in model checking using temporal logic and a finite state model of the behaviour. In our example the properties **NO-CRASH**, and **NO-BLOCK** could be verified with the model checker.

The most difficult case for the verification of requirements specified with temporal logic are infinite (or very large) models. In this case interactive theorem proving is appropriate. Using abstraction techniques allows us to reduce the model, and to produce simple proof obligations. However it is not easy to find the right abstractions that maintain the correctness of the model with respect to the desired property.

Model checking is today the only industrial accepted proof technology. However, without abstraction the application is restricted to the class of small and finite systems. Fortunately many embedded systems belong to this class.

6.2 Refinement

Refinement defines the relation between the requirements specification and the design as well as between the design and the implementation model. The refinement relation [Bro95] is transitive, such that a stepwise development is possible, and monotonic for the composition operators to support a modular development. In this section we present some development steps and the corresponding techniques to ensure correctness.

The easiest refinements are straightforward structural refinements, since they do not require to prove correctness. One example is the refinement of the component Controller in Figure 2 by the structure given in Figure 5. The definition of a behaviour for the components CoreController and Timer is also a refinement

[3] Within project Quest we are evaluating Prolog for this purpose.

step without proof obligation. In the case the refined structure (in our case Controller) has a behaviour associated, it has to be proved that the refined structure or behaviour is indeed a refinement (*behavioural refinement*). In our example the refinement of the states (Figure 7) by the automata describing the substates (Figure 8, and 9) are behavioural refinements.

The proof obligations of behavioural refinement can be proved by model checking (in finite and small cases) or by interactive, deduction based theorem proving (in the general case). Both proof techniques ensure that the behaviour of the refined component is correct with respect to the behaviour specification of the abstract component. Other typical examples are the elimination of underspecification, or the implementation of one data structure by a more efficient one.

Special cases for behavioural refinements are abstractions between two models that occur during the requirements verification, and the refinement of interfaces, which also bases on abstraction techniques [Slo97]. With these development steps every step of the software engineering process can be formally verified, however, since formal refinement is a time consuming activity the effort can be reduced by refining only critical components formally.

In the development process (see Figure 1) there are informal steps from the abstract views towards more concrete views. The formalization by the semantic model and its representation in logics allows us to prove the correctness of such steps. In some situations it might be helpful to do some new refinement steps within the formal model, and to translate the result back to graphical descriptions in order to continue with a semi-formal development. In the project Quest (see Section 7) we provide partial retranslations from formal into graphical description techniques.

6.3 Code Generation

As mentioned in Section 5.3 there exist a generation of code from "executable specifications". One of the crucial challenges in the application of formal methods is to formally prove the correctness of code generators [BBF+92].

6.4 Generation of Tests

In Section 5.4 the general representations for test-cases are described. In this section we focus on the generation of test cases from specifications.

As mentioned in the previous section refinement is quite time consuming and expensive, even compared with producing a formal specification of a system. It might be useful to specify the non-critical components formally, even if no refinement is intended. The reason for this is that a formal specification, even if it is only an abstract interface specification can serve as a good basis for test case generation.

The process of testing is done in the following steps:

1. selection of test cases and input values,

2. determination of output values for all (combinations of) input values,
3. representation of the test (documentation and reuse), and
4. testing the implementation.

There are many methods that allow us to derive test cases from the specification of the components. One popular method is the transition tour which suggests to find a test suite that covers all transitions of the automata describing the behaviour. Problems of this method, developed for automata without input and local variables, are to find the right input values and to ensure that the suggested transitions are possible. We do not go into the details of the test generation here we just refer to the work of the project Quest, especially [Sad98], and show the result of the test cases generation for the timer.

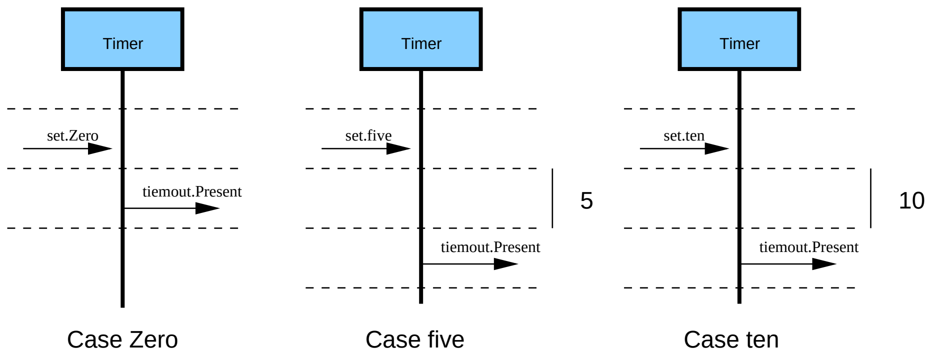

Fig. 10. Test cases for Timer

The input values were selected as a classification of natural numbers. Note that the time-ticks are essential in the diagram, since there is no (observable) signal to express time progress. If the implementation of the timer is deterministic and fulfils our test cases, the correctness of the critical component is ensured.

7 Conclusion and Future Work

We presented an extension of a conventional process model, with integrated formal methods, as far as it is desired, into the software engineering process. The technical foundation for this process model is the uniform semantic basis for the description techniques used within the development process. We demonstrated this technique by an example which we developed with user-oriented description techniques for structure, behaviour, and interaction, based on a model for data types. Our technique of a common semantic basis may also be applied to other description techniques to yield analogous results.

Note that the expressive power of the formal model influences the provable properties. If, for example, real-time properties have to be proved, an adequate model is required, however, a complex model is often more difficult to handle. Finding expressive and practical models is a challenge in research. We decided to use a simple but extendable model for our description techniques that allows us to develop embedded systems and that can be easily extended to more complex applications like real-time, hybrid [MS96], dynamic, and object-oriented [BGH+98] systems.

The biggest part of the effort in applying formal methods is the work in proving the correctness of refinement steps. Since the costs for learning and using formally founded description techniques is relatively small we recommend to use them in many parts of the development for the following reasons:

– Formally founded description techniques are a clear and precise way to model different views of the system.
– It is easier to increase the applications of formal methods within a development project, for example by deciding that a former non-critical component becomes critical.
– Specification based test methods can be applied to generate test cases systematically.

The future work can be divided into two directions. One is the formalization of additional description techniques [BCMR97,Hin98]. However, since many graphical description techniques, especially from UML, have many features, the required semantic model becomes complex [BGH+98] and this will make refinement steps difficult. It is therefore crucial to increase the expressive power of theorem provers and model checkers to handle complex models.

Another direction of future work is to provide tools for the formalization of description techniques with a simple and common semantic basis. In our Munich research group this direction is pursued within the project Quest. The goal of the project Quest is to provide tools for the co-development of embedded systems. To achieve this goal we concentrate on the description techniques presented in this paper, and we connect AutoFocus, the modeling tool with model checkers and the theorem proving environment VSE II [RSW97]. For the development of non-critical parts of the systems we provide a test environment with specification based test generation techniques.

Acknowledgment: For many helpful comments on previous versions of this paper we thank Katharina Spies, Bernhard Schätz, and Peter Braun.

References

[BBF+92] Bettina Buth, Karl-Heinz Buth, Martin Fräzle, Burghard von Karger, Yassine Lakhneche, Hans Langmaack, and Markus Müller-Olm. Provably Correct Compiler Development and Implementation. In Uwe Kastens and Peter Pfahler, editors, *Compiler Construction, 4th International Conference on Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*, pages 141–155, Paderborn, Germany, 5–7 October 1992. Springer.

[BCMR97] Manfred Broy, Derek Coleman, Tom S. E. Maibaum, and Bernhard Rumpe, editors. *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques.* Technische Universitaet Muenchen, TUM-I9803, April 1997.

[BD95] Adolf-Peter Bröhl and Wolfgang Dröschel. *Das V-Modell.* Oldenbourg, 1995.

[BDD$^+$93] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The Design of Distributed Systems: An Introduction to Focus—Revised Version. Technical Report TUM-I9202-2, Technische Universität München, 1993.

[BGH$^+$98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In Martin Schader and Axel Korthaus, editors, *The Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, 1998.

[BHKS97] Manfred Broy, Christoph Hofmann, Ingolf Krüger, and Monika Schmidt. Using extended event traces to describe communication in software architectures. In *Asia-Pacific Software Engineering Conference and International Computer Science Conference, Hong Kong.* IEEE Computer Society, 1997.

[BJ95] Manfred Broy and Stefan Jähnichen, editors. *KORSO: Methods, Languages and Tools for the Construction of Correct Software*, volume 1009 of *Lecture Notes in Computer Science*, New York, N.Y., 1995. Springer-Verlag.

[Bro95] Manfred Broy. Mathematical Models as a Basis of Software Engineering. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 292–306. Springer-Verlag, 1995.

[Bro96] Manfred Broy. Formal Description Techniques - How Formal and Descriptive are they. In R. Gotzhein and J. Bredereke, editors, *FORTE IX, 95-112.* Chapman & Hall, 1996.

[Bro98] Manfred Broy. On the Meaning of Message Sequence Charts. In Lahav, Wolisz, Fischer, and Holz, editors, *1st Workshop on SDL and MSC (SAM98)*, pages 13–32, 1998.

[GSB98] R. Grosu, G. Stefanescu, and M. Broy. Visual Formalisms Revisited. In L. Lavagno and W. Reisig, editors, *CSD '98, International Conference on Application of Concurrency to System Design, Aizu-Wakamatsu City, Fukushima.* IEEE Computer Society Press, 1998.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231 – 274, 1987.

[Hau97] Markus Haubner. Transformation von MSCs in temporallogische Formeln, 1997. Diplomarbeit.

[Hin98] Ursula Hinkel. *Formale, semantische Fundierung und eine darauf abgestützte Verifikationsmethode für SDL.* PhD thesis, Technische Universität München, 1998.

[HMR$^+$98] Franz Huber, Sascha Molterer, Andreas Rausch, Bernhard Schätz, Marc Sihling, and Oscar Slotosch. Tool supported Specification and Simulation of Distributed Systems. In Bernd Krämer, Naoshi Uchihira, Peter Croll, and Stefano Russo, editors, *Proceedings International Symposium on Software Engineering for Parallel and Distributed Systems, pp. 155-164, ISBN 0-8186-8467-4*, pages 155–164. IEEE Computer Society, Los Alamitos, California, 1998.

[Hoa96] C. A. R. Hoare. The Role of Formal Techniques: Past, Current and Future or How Did Software Get so Reliable without Proof? In *18th International Conference on Software Engineering*, pages 233–235, Berlin - Heidelberg - New York, March 1996. Springer.

[HS97] Franz Huber and Bernhard Schätz. Rapid Prototyping with AutoFocus. In A. Wolisz, I. Schieferdecker, and A. Rennoch, editors, *Formale Beschreibungstechniken für verteilte Systeme, GI/ITG Fachgespräch 1997, pp. 343-352*. GMD Verlag (St. Augustin), 1997.

[IT93a] ITU-T. *Recommendation Z.100, Specification and Description Language (SDL)*. ITU, 1993.

[IT93b] ITU-T. *Recommendation Z.120, Message Sequence Chart (MSC)*. ITU, 1993.

[MS96] Olaf Müller and Peter Scholz. Specification of Real-Time and Hybrid Systems in FOCUS. Technical Report TUM-I9627, Technische Univerität München, 1996.

[Par95] D.L. Parnas. Using Mathematical Models in the Inspection of Critical Software. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Applications of Formal Methods*, International Series in Computer Science, chapter 2, pages 17–31. Prentice Hall, 1995.

[RSW97] Georg Rock, Werner Stephan, and Andreas Wolpers. Tool Support for the Compositional Development of Distributed Systems. In *Tagungsband 7. GI/ITG-Fachgespräch Formale Beschreibungstechniken für verteilte Systeme*, number 315 in GMD Studien. GMD, 1997.

[Sad98] S. Sadeghipour. *Testing Cyclic Software Components of Reactive Systems on the Basis of Formal Specifications*. PhD thesis, Technische Universitt Berlin, Fachbereich Informatik, 1998.

[Slo97] Oscar Slotosch. *Refinements in HOLCF: Implementation of Interactive Systems*. PhD thesis, Technische Universität München, 1997.