Consistent Integration of Formal Methods*

Peter Braun, Heiko Lötzbeyer, Bernhard Schätz, and Oscar Slotosch

Institut für Informatik, Technische Universität München
80290 München, Germany

Abstract. The usability of formal concepts for system design depends essentially on their integration in the design process. We discuss several possible levels of integration: *technical* integration of tools considering APIs and tool interfaces, *conceptual* integration of metamodels of description formalisms combined with hard and soft constraints, *semantical* integration of semantics of description techniques using a common semantic model, and finally *methodical* integration by an embedding in the development process. We show the feasibility of such an integrated approach and its advantages presenting AutoFocus/Quest, a formal method CASE-Tool with its levels of integration. Parts of a banking system model are used as example.

1 Introduction

The need for development tools for the design of (embedded) systems has been widely accepted: several programs are available for their construction, often focusing on specific aspects of the design process like building a data-model, verifying system properties, or simulating a designed system. However, generally several of these aspects are important in a thorough design process.

An obvious way to obtain a more powerful tool is to combine existing tools and to integrate them. This approach has also been applied to description techniques like the UML. However, the result of the integration is not necessarily satisfying for the user: There can be redundancies (with the possibility to introduce inconsistencies while modeling overlapping aspects of the system), missing integration of concepts (with the need to bridge a gap between the design and the verification tool), or - as the most critical aspect - no integrated method for the user. These problems arise in conventional software development concepts as well as in formal method approaches.

In this paper we advocate a new multi-level integration concept. The most sophisticated level is the methodical integration, ideally based on the next level, the semantic integration of the used description techniques. The third level forms the conceptual integration of metamodels, followed by the lowest integration level, the technical integration of tools. Integration on all levels results into powerful

* This work was supported by the Bundesamt für Sicherheit im Informationswesen (BSI) within the project Quest, and the DFG within the Sonderforschungsbereich 342.

S. Graf and M. Schwartzbach (Eds.): TACAS/ETAPS 2000, LNCS 1785, pp. 48–62, 2000.

tools that provide a lot of features to the user. The layered integration hierarchy makes it easy to connect other programs, provided that the development methods fit together.

As an example for this new integration hierarchy we present the tool AutoFocus/Quest, offering many different features, ranging from different graphical description techniques to theorem proving, testing, code generation and model checking. AutoFocus/Quest integrates the CASE tool prototype AutoFocus with the formal tools VSE II, SMV, SATO, and CTE. This paper is structured as follows: after a short overview over the tool AutoFocus/Quest in Section 2, we present (parts of) the banking system of the FM99 tool competition in Section 3 to introduce our description techniques. The main part of this paper (Section 4) describes the different levels of integration that are present in the AutoFocus/Quest tool. We conclude with a comparison with other existing tools.

2 The AutoFocus/Quest-Tool

AutoFocus/Quest has been presented successfully at the Formal Method World Congress 1999[1]. In the following we briefly describe the features of the tool.

2.1 AutoFocus

AutoFocus [HMS+98] is a freely available CASE-Tool prototype for the development of correct embedded systems. Similar to other CASE-Tools it supports graphical description of the developed system using several different views. AutoFocus builds upon formal methods concepts. The available views are:

- Interface and structure view: By using System Structure Diagrams (SSDs) users define structure and components of the developed system and the interfaces between components and the environment.
- Behavior view: State Transition Diagrams (STDs) describe the behavior of a component in the system.
- Interaction view: Extended Event Traces (EETs) capture the dynamic interactions between components (and the environment). EETs are used to specify test cases or example runs of the systems and have a Message Sequence Chart-like notation.
- Data view: (textual) Data Type Definitions (DTDs) define the data types for the description of structure, behavior and interaction diagrams. We use functional datatypes.

All views are hierarchic to support descriptions at different levels of detail. AutoFocus can check the consistency between different views using an integrated consistency mechanism. AutoFocus offers a simulation facility to validate the specifications based on rapid prototyping.

[1] AutoFocus/Quest was acknowledged as the leading competitor in the tool competition of FM'99. See `http://www.fmse.cs.reading.ac.uk/fm99/` for more information.

2.2 Quest

Within the project Quest several extensions to AutoFocus were achieved.[2] The aim of the project [Slo98] was to enrich the practical software development process by the coupling existing formal methods and tools to AutoFocus in order to ensure the correctness of critical parts of systems. The combination with traditional software engineering methods is achieved by specification-based test methods which facilitate the selection of reasonable test-cases for non-critical components. The tools developed within Quest translate the graphical concepts into other formalisms suitable for verification, model checking or testing (VSE [RSW97], SMV [McM92], SATO [Zha97], CTE [GWG95]) and support the systematic generation of test-cases. To provide an integrated, iterative development process (partial) retranslations from the connected tools were implemented, supporting visualization of counterexamples, generation of test sequences with input values, and the reimport of components that have been corrected during verification. The translations were realized using an API and a textual interface to AutoFocus. This allows to easily connect other tools using interfaces or generators.

3 The FM99-Banking System

In this section we briefly present parts of a model that was presented on the Formal Methods World Conference 1999.[3] The banking system example was used to compare different modeling methods and verification tools, presented at FM'99, on a competitive basis.

The banking system contains a central (main host) and several tills (automated teller machines) that are communicating independently with the central database. The connections from the central to the tills may be down. Among other requirements it has to be ensured that the amount withdrawn within one day does not exceed a maximum value.

We modeled the banking system with the graphical description techniques from AutoFocus/Quest. In the following sections we describe the main structure of the system, some of the datatypes used, the behavior of the connection, and a possible interaction sequence from a connection and the central.

3.1 System Structure

We modeled the system with two tills and two connections. Since the connections have a specific behavior (i.e. they can be down), we modelled connections as components in the system. The structure of the system is described in the Fig. 1. The connections pass transactions (of type `Info`) from the tills to the central, and in the other direction connections report information to control the behavior of

[2] The project was carried out for the German "Bundesamt für Sicherheit in der Informationstechnik" (Information Security Agency) (BSI).

[3] See `http://www4.in.tum.de/proj/quest/` for the full model.

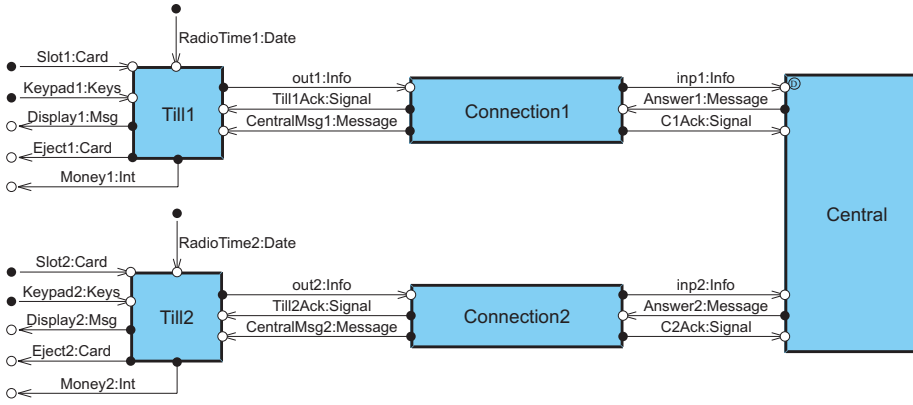

Fig. 1. SSD for Banking System

the till. The messages (of type `Message`) are defined in a DTD (see Section 3.2). Furthermore the connections have acknowledge channels to indicate the sender of a message that it has been sent successfully. The SSD shows the ports (little circles) of the components, the external ports, and the channels that connect these ports.

Reusing port names[4] in different components allows us to describe the behavior of each connection with the same state transition diagram. For example the channel `out1` connects to the port `out` of `Connection1`.

3.2 Datatypes

As simple examples, we show the definition of two datatypes and a function (defined using pattern matching) used within the graphical views:

```
data Message = Money(Int) | NoMoney | Balance | MailSent;
data Transaction = TA(Action,Account);
fun withdrawMoney(TA(Withdraw(x),acc))=Money(x);
```

3.3 Behavior

The behavioral description of the system refers to the ports of the components defined by the SSD in Section 3.1. The behavior of both connections are described in Fig. 2. If there are no values on both channels (expressed by the input patterns: `out?;Answer?`) the connection changes its state to reduce energy usage. This has been introduced to model the fact that connections sometime are down. If the connection is down and receives a value it moves to the state sleeping. If another value is received, then the connection is up, and ready to process the data.

[4] Port names are attributes of ports.

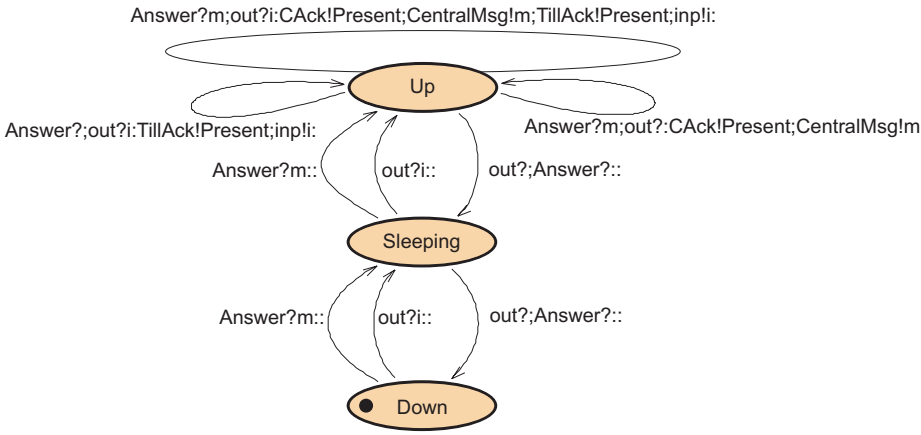

Fig. 2. STD for `Connection1`

3.4 Interaction Example

To illustrate the interaction from the connections and the central we use the EET in Fig. 3. This is only one example trace of the behavior of the system. All messages between two ticks (dashed lines) are considered to occur simultaneously.

4 Integration

In this section we describe the integrations within AutoFocus/Quest. There are different integration layers:

- methodical integration (on the development process level)
- semantic integration (on the semantics level)
- conceptual integration (on the metamodel level)
- technical integration (on the tool level)

The quality of an integration depends on the reached level, and on the quality of the underlying levels, for example a complete semantic integration is not sufficiently helpful for a user without tool support.

4.1 Metamodel Integration

In our view, integrated metamodels are essential for further integration. Metamodels can be applied to define a modeling language [Met99] and are increasingly used, for instance to define UML [BJR97]. In the case of AutoFocus the syntactic aspects of the modeling techniques, like SSDs and STDs, are described using a metamodel. We use class diagrams as defined by the UML for the description

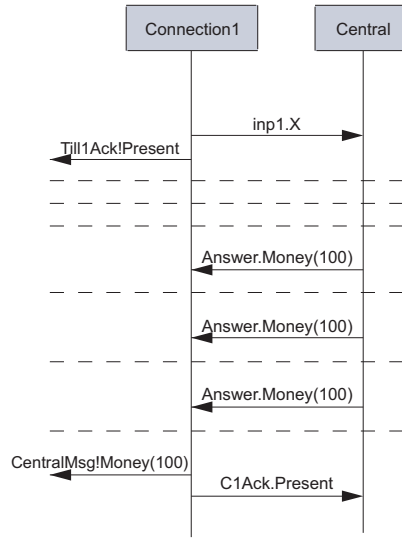

Fig. 3. Execution Example

of metamodels. We consider it essential that the metamodels of different concepts of a modeling language are integrated into one metamodel since otherwise different modeling concepts do not fit together. We show this for the example of SSDs and STDs.

Fig. 4 shows a simplified version of the integrated metamodel for SSDs and STDs. An SSD consists of components, ports, channels and relations between them. A component has a name, and can be decomposed in an arbitrary number of subcomponents, and can belong to a supercomponent. This is needed to model hierarchic components. A component is a composition of ports and channels. In the example of Fig. 1 the component `Connection1` has four output ports and two input ports and no channels. All channels in Fig. 1 belong to the component `Banking System`, the supercomponent of the shown components. Below the metamodel of SSDs is the metamodel of STDs. An automaton consists of one state, which may be hierarchic, as the component in the STD metamodel. Similar to the component a state has interface points and transition segments. A transition segment is a composition of a precondition, some input patterns, some actions and output patterns. See the example of Section 3 for an explanation of those elements.

We now have two concepts, the concept of system structure diagrams and the concept of automatons. But how do they fit together? In many other tools there will be only an association from components to automata: the dynamic behavior of a component can be described with an automaton. But there is more. In our case the input and output patterns that belong to a transition segment are connected to the ports of the component to which the automaton

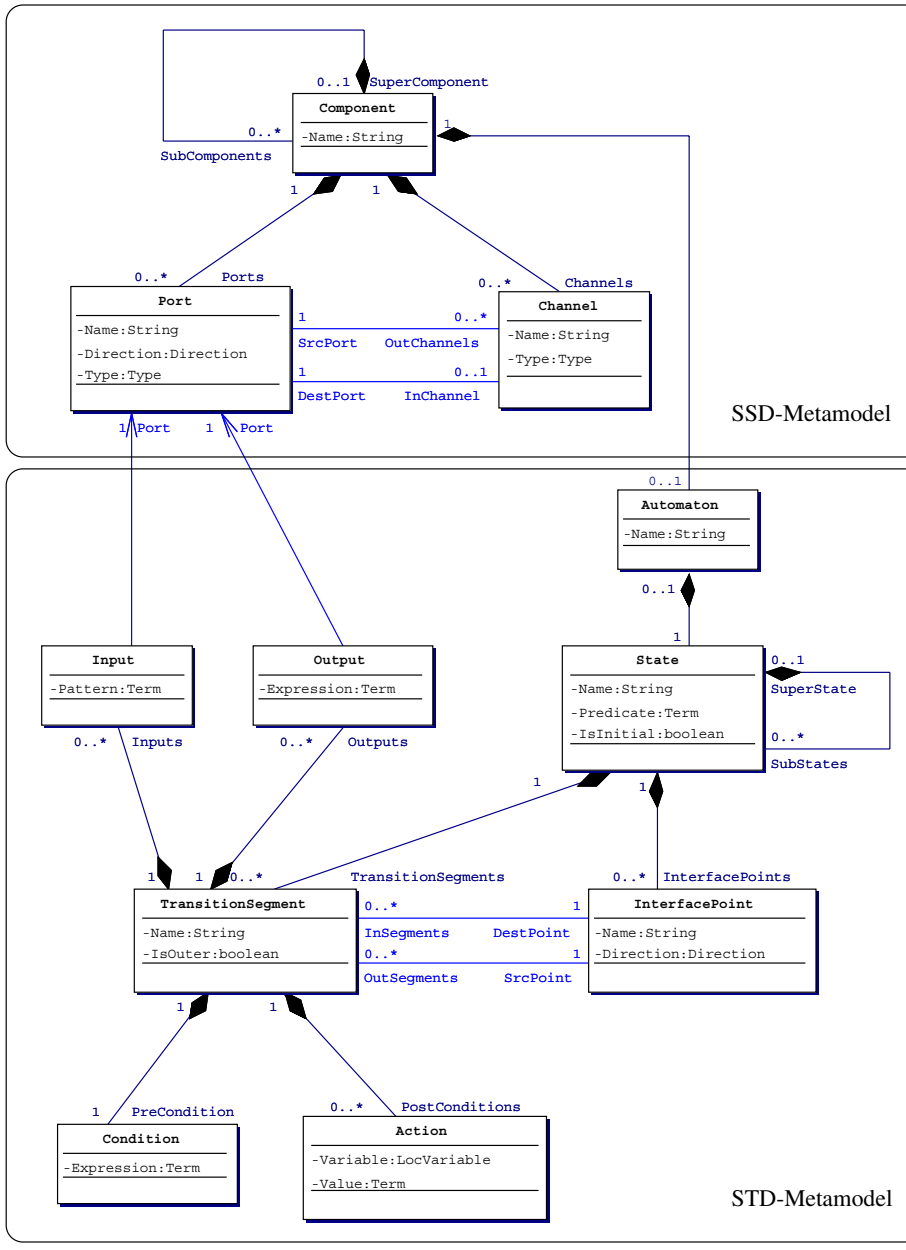

Fig. 4. Integrated Metamodel for SSDs and STDs

belongs. So an input to the automaton of component `Connection1` in Fig. 1 can only come from the ports `Answer` or `out` restricting what is considered a correct model. Thus for specifying a new input pattern one might want to only choose from ports of the related component or to create a new port, which then automatically belongs to the related component. Note that the constraint, that a port belongs to the right component, is not modelled in the class diagram of Fig. 4. There, only the fact is expressed, that every input has a relation to an arbitrary port. The constraint that the automaton belongs to the same component as the ports of the input and output patterns, is expressed in a logical constraint that belongs to the metamodel. Up to now AutoFocus uses its own syntax for these expressions. However, a prototype exists using OCL [WK98], the constraint logic of UML [BJR97].

Beside the integration of the concepts of SSDs and STDs there are other concepts integrated in AutoFocus/Quest. For instance, datatypes are modelled and used. In Fig. 4 attributes with types like term or type are shown. These are actually relations, like the ones from input and output to port, to classes in other parts of the integrated metamodel. So we do not only have different concepts, but we have merged them tightly together. But sometimes the user is handicapped by a very strict link between the different views of a system. So we tolerate the possibility that the model might be inconsistent at some stages of the development. For example, when specifying an SSD, it is sometimes desirable that datatypes can be used that are not yet defined. Thus, especially for the links between different concepts, we tolerate inconsistencies. Hard constraints like the need for a channel to have relations to a source-port and a destination-port may not be violated.

4.2 Semantic Integration

As with the conceptual level, where all system views are mapped onto a common metamodel, all description techniques are also mapped onto one semantic model. Thus, all descriptions are semantically integrated on the mathematical or the model level as sketched in the following subsection. However, this is a very low level of integration not suitable for system development. Instead we need to express the semantics of the described systems in terms of the techniques and actions performed by the user. The highest form of integration is achieved if the user can completely stay within the level of formalization used in the specification process so far, for instance, to SSDs, STDs, EETs, and DTDs, as described in the second subsection. However, this is not generally possible and thus other forms of integration adding new formalisms or techniques must be used as described in the last subsection.

Semantic Model As already suggested by the STD description formalism, we use a stepwise computation mechanism to model the behavior of a system or component. Each step consists of two substeps: reading the input ports of a component and processing the input, and generating the output. After each step

the output of a component is transferred from its output ports along its channels to the input ports of the corresponding channels. The behavior of a component or a system is formalized by sequences of those steps.

One of the major incentives in choosing this semantic model is its simplicity. Furthermore, it is compliant with other approaches – both from the engineering and the formal methods domain – that are widely accepted. The concept of stepwise, cyclic computations with unbuffered variables (channels) is, for example found, in programming languages for PLCs (programable logical controllers) as well as in TLA (temporal logic of actions). This straight-forward yet expressive semantic model seems to be best suited for the domain of hardware oriented embedded systems compared to approaches using more complex concepts, for example state charts with the concept of OR- and AND-states. Nevertheless, other semantic models can be adapted to fit the basic principles of AutoFocus/Quest, adding concepts like buffering channels suitable for other domains like telecommunication.

Consistent Specifications As already mentioned on the conceptual level of the metamodel, during a development process inconsistencies in system specifications may occur. This is especially the case using a view-based approach where the specification is simultaneously presented on different levels of abstraction and spread over different aspects like structure (SSDs), behavior (STDs) or interactions (EETs). Then mechanisms must be offered to support the user in finding those inconsistencies. Those inconsistencies may occur on the conceptual level, for example if the type of a port does not meet the value sent on it. On the conceptual level those checks are comparably simple and can be carried out automatically. However, even in the semantical level different inconsistencies may occur:

Abstract vs. concrete behavior: As mentioned in section 4.1, a component (or system) may be realized by a number of subcomponents using an SSD. Accordingly, the system designer can assign behavior to both the component and its subcomponents using STDs. In the AutoFocus/Quest development methodology the assignment of behavior to the 'black box' and 'glass box' view of a system is considered as a refinement step performed by the designer. Thus, it should be checked that the abstract component including its behavior is refined by the concrete subcomponents and their behavior given by their STDs. Since this is a refinement step, the concrete behavior must fulfill the requirements of the abstract behavior, or - in other words - any behavior exhibited by the concrete system must also be possible in the abstract version.

Behavior vs. interaction: As mentioned in section 3 the views used in the AutoFocus/Quest approach are not completely independent and may share common aspects. On the semantic level, both SSDs combined with STDs and EETs express behavior of a system (or component).

Again, support is needed to aid the user during the development process in detecting such inconsistencies. Again, those checks should be performed as au-

tomatic as possible. If the state space of the specified system is finite, in general model checking can be used to perform those checks. We used the model checker μcke [Bie97] to implement a prototype of such an automatic check ([SH99], [Bec99]. Naturally, the performability of such checks is heavily influenced by the size and the complexity of such a system. For the second check, i.e., verifying that an interaction described by EET corresponds with the behavior of a system characterized by SSDs and STDs, another proof method can be applied using a bounded model checking approach. Here, the model checker SATO is used to check that form of consistency (see [Wim00]).

Further Integration As mentioned above, automatic checks can not always be performed. In that case, other forms of integration must used.

One possible solution is to exchange the model checker by a theorem prover, as it was done with AutoFocus/Quest using VSE. This integration adds a new description formalism, the logical formalism of the theorem prover, to the existing set of description techniques. Thus, this approach requires the user to get familiar with another formalism as well as an additional tool, the prover. While this results in a weaker integration from the user's point of view, it extends consistency checks to more general systems with an infinite state space.

Another possibility is to add another description formalism to express semantic properties of a component or system and integrate a suitable tool for this formalism. In AutoFocus/Quest both SATO (see [Wim00]), and SMV (see [PS99]) were integrated. This approach adds a new description formalism, the temporal logic, and does not treat the inconsistencies mentioned in the previous subsection. However, it offers two advantages for easy integration in the tool environment:

- The checks can be performed automatically, thus - from the user's point of view - there is no need for another tool.
- The check either passes or results in a counter example expressed in form of an EET. Thus, besides the temporal logics, no other formalism is needed.

4.3 Methodical Integration

The last sections dealt with the integration of the semantical concepts behind the conceptual layer. Integrated and consistent semantics are not only a well founded basis but also a prerequisite for a continuous and straight-forward software development process. As a prime requisite to up-to-date system development, the AutoFocus/Quest method covers the development process from the graphical description of the system over system verification using theorem proving to code generation. Beyond that, however, the methodical integration of different software development concepts is essential for a successful tool integration. An adequate integration of different methods will make any integrational tool worth more than the single parts from which it was made.

Generally, the development process is divided into phases like requirements, modeling, validation, prototyping, and test. In our approach, while each phase

may use different techniques, they all work on the same metamodel with a common semantics. This enables the developer to switch between different phases like specification, verification, or testing, and the corresponding tasks without the need for complex manual transformations. Due to the versatility of the model based approach, there are no restrictions introduced by the model which complicate the combination of those tasks or integration of new methods in the process. The next paragraphs discuss the different phases in detail and show how they fit into the integrated metamodel and the given semantics.

The **requirements phase** generally deals with informal documents and fuzzy specifications. We provide extended event traces (EET, see Section 2.1) that allow the specification of use cases as exemplary interaction sequences. EETs, an AutoFocus description technique, represent a first, generally incomplete, specification of the system.

In the next phase, the **modeling phase**, the skeleton of the system specification is enlarged and refined to get a more defined structure and behavior model. This is done by applying the AutoFocus development method (See [HMS+98]). If one or more EETs already exist, it is also possible to derive the first version of the interface and the system structure from the axes and messages of the EETs.

In the **validation phase** the following tasks are supported:

- consistency checks (see Section 4.1),
- simulation (see [HMS+98]),
- model checking (see [PS99]), bounded model checking (see [Wim00]) and
- theorem proving.

AutoFocus/Quest provides simulation of components. So, faults in the specifications can be detected and located directly while the simulation is running. Beyond that, the simulation also generates system traces which are displayed as EETs and can be later used for conformance testing.

In case of developing safety critical systems, validation by simulation is not sufficient. Often formal proofs are required to show the correctness of the specifications. With model checking it is possible to prove safety critical properties of components automatically. Therefore model checking became very popular. Model checking does not only check properties as valid, but also produces counter examples if the property does not hold. If the model checker finds a counter example, it is retranslated and displayed as an EET. Abstraction techniques are a further example for a method integration. They extend the power of model checking to large and infinite systems (we use [Mül98]). Simple proof obligations are generated (for VSE) to ensure that the model checked property in the abstract system also holds in the concrete systems. The design task, to find correct abstractions, is supported within AutoFocus/Quest.

Besides the model checking approach, AutoFocus/Quest also allows to translate the specification to a representation understood by the VSE system, a theorem prover for a temporal logic similar to TLA supporting hierarchic components like AutoFocus/Quest [RSW97]. Within the VSE system a formal founded validation of the specification can be performed. It is also possible to

make small changes of the specification in the VSE system and the re-import to AutoFocus/Quest.

Beside system validation AutoFocus/Quest supports the **test phase**. Testing in AutoFocus/Quest focuses on conformance testing between a specification and its manually coded implementation. The most challenging task within software testing is a tool-supported test case generation. Therefore we have integrated the classification tree editor CTE [GWG95]. The CTE allows the classification of arbitrary datatypes with trees and the selection of certain combinations as test input classes. The construction of the classification trees is assisted by the classification tree assistant (CTA) which can be set up individually. In AutoFocus/Quest we generate the CTA from datatype definitions, system structure, and automata. So the tester does not need to start from scratch, but can use the suggested standard classifications. The classification tree method is combined with a sophisticated test sequentialization which computes test cases, i.e. sequences of input/output data. Test cases are coded as EETs. A simple test driver allows the execution of the generated test cases with Java components.

The **implementation phase** is supported with Java and C code generators.

A well done methodical integration of different software development methods does not restrict the development process but gives more freedom. In AutoFocus/Quest EETs play an important role in integration. EETs are a kind of multi-functional description technique. They serve as use cases, interaction description, counter examples, as well as test cases.

4.4 Tool Integration

In the previous section we have shown the methodical integration of the development process. Through the different phases of the development process we have to use different tools, like AutoFocus, model checkers, theorem provers and test tools. So we designed AutoFocus/Quest not to be one monolithic tool, but rather to be a collection of different specialized tools which are based on the same integrated metamodel. One big advantage of having many different tools is, that they are small, easier to understand and to replace. One example for this is the connection to the model checkers SMV and SATO. You can use each one or both for AutoFocus/Quest or we can build another connection to a new model checker, which replaces the other two connections.

To link our tool-collection together we use a central repository with a conceptual schema, which is defined by the integrated metamodel. Every other tool can use the repository via an API. The API offers methods for e.g. importing or exporting a complete repository to a text file or to do some consistency checks on a model or a part of it. Note that the consistency checker bases on the core API, so that different tools to check the consistency of our models can be used.

Since we would like to integrate other tools, we want to have a flexible metamodel, and we want to be able to change things in the metamodel or to add new concepts to the metamodel without having to recode the central repository and without the need to touch every tool even if it is not affected by the change. So the core repository, which offers an API to create, modify or delete objects, is

completely generated. As the core API the textual importer and exporter are generated [Mar98]. Thus we only have to change tools which are directly affected by the change. The generated core repository also ensures that our hard constraints are not violated. Every programmer using the API can rely on the fact that the hard constraints are not violated in any model, making programming considerably easier.

A common, well documented model is also important for integration of new tools. To connect a new tool to AutoFocus/Quest directly the repository API or a translation from the model data in the repository to the data needed by the tool can be used. Consistency checks can be used to ensure some constraints that are a prerequisite for the translation. Besides this a retranslation of results or model changes to AutoFocus/Quest might be needed.

The spectrum of currently available tools in the field of embedded systems ranges from those tool focused on formal based design including verification techniques to tools concentrating on software engineering aspects including view-based system description, code generation and simulation. Consider, for example, Atelier B [Abr96] on the one hand, StateMate [Har90] or SDT [Tel96] on the other hand. Generally, those tools only partially integrate those aspects. An integrated combination of clearly defined but simple modeling concepts, an underlying well-defined semantic model for verification purposes, industrial-oriented notations supporting view-based development, prototyping for requirements validation, code generation for system implementation as well as test cases generation for system validation.

On the SW-engineering side, this is often due the fact that system modeling approaches like UML [BJR97] using different, graphical notation for the description of systems lack a sound conceptual and semantical basis. As a consequence, in many cases tools like Rational Rose [Rat98] and Rhapsody [i-L97] supporting this notation are missing a suitable formal semantics. Thus, while those notations and tools do offer support for checking syntactic consistency conditions, no such support is available on the semantic side. Those observations even hold for notations and tools based on more formally defined approaches like SDL and corresponding tools like SDT [Tel96].

On the other side, tools originating in formal approaches like STeP [BBC+95] or Atelier B [Abr96] often lack methodical or engineering functionality like the splitting of complete system descriptions into well-defined views of the system, combining these descriptions on different levels of abstraction as well as relating those views to form a consistent description.

The goal for developing AutoFocus was not to build another tool but rather to investigate how the experiences gained in several basic research projects could be consequently applied to the development of embedded systems. Some aspects were considered as major topics: the use of well-defined description techniques, the modularity and expressiveness of those description techniques supporting different levels of abstractions and view-points, as well as the methodical integration of those techniques for integrated development approach. Those aspects also are the features distinguishing AutoFocus from other tools in this area.

For example, approaches like StateMate mix different views (system structure and behavior by using AND- and OR-states). Furthermore, there a semantical basis is not applied to support model checking or theorem proofing. While other tools avoid those mixing of aspects, they are somewhat lax in the use of complete and clearly defined description techniques. Thus, for example, in ObjecTime, the diagrams for behavioral description are annotated with program code fragments to for a complete description. Since ROOM [SGW94] and ObjecTime were developed without a clear semantical model, in the end the meaning of diagrams is only described by the generated executable models, leaving the possibility for open questions concerning modeling concepts as well as lacking the requirements for formal support by theorem proofing or model checking.

The combination of the above mentioned aspects and the resulting methodical consequences are central incentives for the development of AutoFocus. AutoFocus supports a lean subset of description techniques based on a common mathematical model. These description techniques are independent of a specific method or a tool, while offering the essential aspects of similar description techniques, and can therefore be combined with a wide rage of methods and development processes.

5 Conclusion and Future Work

Integration of formal techniques is more than just integrating formal (and semiformal) tools; nevertheless, tool integration is one important step. Several tool platforms that can be readily connected to AutoFocus/Quest, including NuSMV, Proovers, STeP, or Isabelle. Since the ultimate goal is the methodical integration, more steps have to be taken on this level: Code generators for C and Java are currently under development. Further work is needed in requirements phase, to support methods for requirements tracing (as found in the DOORS tool). Furthermore, in the modeling phase the use of graphical support for model based development steps (refinement, splitting of transitions, combining channels, etc.) in AutoFocus must be investigated.

References

Abr96. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996. 60

BBC+95. Nikolaj Bjorner, Anca Browne, Eddie Chang, Michael Colon, Arjun Kapur, Zohar Manna, Henny B. Sipma, and Tomas E. Uribe. STeP: The Stanford Temporal Prover (Educational Release) User's Manual. STAN-CS-TR 95-1562, Computer Science Department Stanford University, 1995. 60

Bec99. Roland Bechtel. Einbettung des μ-Kalkül Model Checkers μ-cke in AutoFocus. Master's thesis, Institut für Informatik, Technische Universität München, 1999. 57

Bie97. Armin Biere. *Effiziente Modellprüfung des μ-Kalküls mit binären Entscheidungdiagrammen.* PhD thesis, Universität Karlsruhe, 1997. 57

BJR97. G. Booch, I. Jacobson, and J. Rumbaugh. *UML Summary*. Rational Software Cooperation, January 1997. Version 1.0. 52, 55, 60

GWG95. M. Grochtmann, J. Wegner, and K. Grimm. Test Case Design Using Classification Trees and the Classification-Tree Editor. In *Proceedings of 8th International Quality Week, San Francisco*, pages Paper 4–A–4, May 30-June 2 1995. 50, 59

Har90. D. Harel. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990. 60

HMS+98. F. Huber, S. Molterer, B. Schätz, O. Slotosch, and A. Vilbig. Traffic Lights - An AutoFocus Case Study. In *1998 International Conference on Application of Concurrency to System Design*, pages 282–294. IEEE Computer Society, 1998. 49, 58

i-L97. i-Logix. *Rhapsody Reference Version 1.0*, 1997. 60

Mar98. Frank Marschall. Konzeption und Realisierung einer generischen Schnittstelle für metamodell-basierte Werkzeuge. Master's thesis, Institut für Informatik, Technische Universität München, 1998. 60

McM92. K.L. McMillan. The SMV system, Symbolic Model Checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992. 50

Met99. MetaModel. http://www.MetaModel.com/, 1999. 52

Mül98. Olaf Müller. *A Verification Environment for I/O-Automata Based on Formalized Meta-Theory*. PhD thesis, Institut für Informatik, Techn. Univ. München, 1998. 58

PS99. J. Philipps and O. Slotosch. The Quest for Correct Systems: Model Checking of Diagramms and Datatypes. In *Asia Pacific Software Engineering Conference 1999*, pages 449–458, 1999. 57, 58

Rat98. Rational. Rational Rose 98 Product Overview. http://www.rational.com/products/rose/, 1998. 60

RSW97. G. Rock, W. Stephan, and A. Wolpers. Tool Support for the Compositional Development of Distributed Systems. In *Proc. Formale Beschreibungstechniken für verteilte Systeme,GI/ITG-Fachgespräch*. GMD-Studien Nr. 315, ISBN: 3-88457-514-2, 1997. 50, 58

SGW94. Bran Selic, Garth Gullekson, and Paul Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, 1994. 61

SH99. Bernhard Schätz and Franz Huber. Integrating Formal Description Techniques. In Jeanette Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 - Formal Methods*, pages 1206–1225. Springer, 1999. 57

Slo98. O. Slotosch. Quest: Overview over the Project. In D. Hutter, W. Stephan, P Traverso, and M. Ullmann, editors, *Applied Formal Methods - FM-Trends 98*, pages 346–350. Springer LNCS 1641, 1998. 50

Tel96. Telelogic AB. *Telelogic AB: SDT 3.1 Reference Manual*, 1996. 60

Wim00. G. Wimmel. Using SATO for the Generation of Input Values for Test Sequences. Master's thesis, Technische Universität München, 2000. 57, 58

WK98. Jos Warmer and Anneke Kleppe. *The Object Constraint Language*. Addison-Wesley, 1998. 55

Zha97. H. Zhang. SATO: An efficient propositional prover. In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *LNAI*, pages 272–275, Berlin, July 13–17 1997. Springer. 50