Tool supported Specification and Simulation of Distributed Systems*

Franz Huber, Sascha Molterer, Andreas Rausch, Bernhard Schätz, Marc Sihling, Oscar Slotosch
Institut für Informatik
Technische Universität München
80290 München, Germany
(huberf|molterer|rausch|schaetz|slotosch|sihling)@informatik.tu-muenchen.de

Abstract

We present prominent features of AUTOFOCUS*, a tool prototype for the formally based development of reactive systems.* AUTOFOCUS *supports system development offering integrated, comprehensive and mainly graphical description techniques to specify both different views and different levels of abstraction of the system. To avoid ill-defined specifications, consistency conditions on these system descriptions can be formulated and checked. Furthermore, we show how consistent and executable specifications of systems or components can be animated using the Java code generation of the* AUTOFOCUS *simulator. Finally, we demonstrate how* AUTOFOCUS *can be used to simulate the specified system in its system environment using a graphic animation tool as an example.*

1. Introduction

Smaller hardware and growing systems are leading to a new degree of system complexity. An additional source of complexity is introduced in distributed systems by communication. With this increasing complexity the number of errors will increase unless methods and tools to properly design those systems are provided.

Formal based methods and tools [4] enable us to prove the correctness of a program with respect to a specification. Since proving is a laborious and expensive task, the specification should capture all relevant aspects for verifying safety critical properties. Furthermore, since not all aspects of real-world sized examples are relevant, the specification and the verification should focus only on the important parts. Finding an ideal level of abstraction requires two features of the applied formalism: It should offer different integrated view points of a system to describe all relevant aspects, and support abstraction and modularisation techniques to allow specification at the right level of detail.

However, the correctness of the initial requirement specification cannot be proved. To validate the adequacy of a specification a simulation component is very helpful. If the simulator is based on code generation, the generated code can be used for prototyping as well. Another useful feature, supporting the acceptance of formally founded methods and tools, is a graphical interface hiding the mathematical expressions denoting the systems semantics as far as possible.

1.1. Related work

Obviously, tool support for the development of distributed systems is not new. The spectrum ranges from tools for mainly verification oriented approaches to those focusing on simulation and code-generation, like [1] or [6] and [12]. In general, however, these apporaches do not try to combine the aspects of intuitive description techniques, a strong semantical basis and verification support as well as simulation and code generation.

For example, the UML method [2] includes several description techniques and offers tool support for the specification of systems using those techniques. However, UML lacks a precise semantical base and therefore neither allows to check the consistency between different parts of the description nor supports the verifivation of system properties.

AUTOFOCUS supports a lean subset of description techniques based on a common mathematical model. These description techniques are independent from 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.

*The authors of this paper were founded by the DFG Sonderforschungsbereich 342 “Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen”, the project KORSYS financed by the German Federal Ministry of Education and Research (BMBF) and the “Forschungsverbund Software Engineering” (FORSOFT).

1.2. Structure of this paper

The aim of this paper is to give an overview over the current state of the AUTOFOCUS tool, including the new features of animation and simulation of specifications.

Section 2 gives a brief sketch of the semantical basis of AUTOFOCUS. The semantics allows us to define consistency criteria between parts of the specifications (see 2.6), and to formally reason about the specified systems.

Section 3 describes SimCenter, a simulation environment integrated into AUTOFOCUS. SIMCENTER is based on generating an executable prototype from consistent specifications and observing its behavior, which allows validation of system properties.

Finally, Section 4 concludes the paper and describes ongoing and future activities of the AUTOFOCUS group.

For the technical details of code generation and consistency, we refer to [8] and [9], respectively. A case study [7] describes a complete development of an distibuted system example with AUTOFOCUS.

2. Specification of Distributed Systems

Only the description from several points of view forms a comprehensive and structured picture of the system. Therefore, AUTOFOCUS offers four different description techniques:

- system structure diagrams (SSDs),
- data type definitions (DTDs),
- state transition diagrams (STDs), and
- extended event traces (EETs),

each one covering different views on the system. The integration of the views on a common semantic basis leads to one integrated formal system specification of the system.

AUTOFOCUS supports component-oriented development of systems. A component is a structural part of the system, possibly described by different views using different description techniques. With SSDs systems can be hierarchically structured into components, described by other hierarchical views. Depending on the granularity, components or views can be atomic, or consist of sub-components or sub-views themselves. Therefore, AUTOFOCUS allows the user to switch between levels of different granularity by using the hierarchical description techniques described in the following sections.

2.1. System Structure Diagrams (SSDs)

A (distributed) system consists of its components and the communication channels among them. An embedded system communicates with its environment. To describe the static aspects of distributed systems, viewing it as a network of interconnected components with the ability to exchange messages over channels, we use system structure diagrams.

Each component has a name and a set of input and output ports to which the communication channels are attached. Every channel is defined by a channel name and a data type describing the set of messages that may be sent on it. In case no value is sent, the channel has a default nil-value. Thus system structure diagrams provide both the topological view of a distributed system and the signature (the syntactic interface) of each individual component:

In AUTOFOCUS components in a SSD can be associated with

- substructures (SSDs),
- other views (STDs or EETs), and
- local component data declarations.

A component data declaration declares local variables for the component, by assigning a name and a data type.

Graphically, SSDs are similar to data flow diagrams, represented as graphs, where rectangular nodes symbolize components and arrow-shaped edges stand for channels.

2.2. Datatype Definitions (DTDs)

The types of the data processed by a distributed system are defined in a textual notation. We use the basic types and data type constructors as for example found in functional programming languages like Gofer [10][1]. The data types defined here may be referenced by other views, for example as channel data types in SSDs, or by local variables of components.

2.3. State Transition Diagrams (STDs)

State transition diagrams are used to describe dynamic aspects, i.e. the behavior of a distributed system and of its components. STDs are extended finite automata similar to those introduced in [5].

Each system component of an SSD can be associated with an STD and each state of a STD can have a STD as substructure. Each transition may have the following annotations:

- pre- and postconditions, formulated over the local data state of the component,

[1] In the actual implementation of AUTOFOCUS we use Java data types because these types can be directly used within the simulated Java code. Since we prefer the more algebraic style of functional data types for specifications of distributed systems, we are implementing a translation from a Gofer subset into Java.

- a set of input and output patterns describing the messages that are read from or written to the input and output channels of the component, and
- a informal label which is displayed, if present, reducing the amount of annotations in the diagram.

Graphically, automata are represented as graphs with labeled oval nodes as states and arrows as transitions.

The treatment of substructures in STDs and SSDs is similar: for every edge (transition/channel) into or from the node (component/state) a port is created in the substructure. This port can be connected by edges from inside the substructures. Ports have no formal semantics. They are just interfaces between different levels of abstraction, and help the system developer to develop the specifications in a modular way. Furthermore, ports can be used to formulate syntactic consistency conditions (see 2.6).

2.4. Extended Event Traces (EETs)

Besides STDs, extended event traces may be also used to describe behavior of distributed systems by exemplary runs from a component-based view. We use a notation similar to the ITU- standardized message sequence charts (MSCs) with core concepts taken from [13]. As well as other graphical AUTOFOCUS notations, EETs support hierarchical concepts. Using "boxes" a number of sub-EETs can be grouped together to specify variants of behavior in parts of an EET. Additionally, indicators can be used to define optional or repeatable (sub)parts of an EET. A complete description of EETs can be found in [11].

EETs can be used at different development stages with different purposes:

- in early stages of system development to specify elementary functionality or error cases by examples,
- later in the development process, the system specifications given by SSDs, STDs, and DTDs can be checked against the EETs, whether the system fulfills the properties specified in them, and
- during validation, EETs can be used to visualize simulation results or error paths, obtained from model checking the system.

2.5. Description Techniques Semantics

While the meaning of DTDs, and SSDs is quite obviou, the exact meaning of the behavioral description techniques STDs was left open so far. In the following subsection we will give a short description of the meaning of STDs. For a more complete description see [9].

For the AUTOFOCUS description techniques, different but coherent semantic models are used. Since AUTOFOCUS was developed based on work from the FOCUS project, stream-based semantics as described in [3] are used. Furthermore, a relational μ calculus semantics was designed compliant with the above semantics. They will be used for verification purposes and model checking, respectively, as mentioned in Section 4.

2.5.1. Meaning of STDs

An STD characterizes the behavior of a system or component reacting on input received from its environment and producing output sent to the environment. Reactions depend on the actual state of the component and influence the future behavior by setting a new state. Since an STD describes an extended finite state machine using variables local to the characterized component, the state of a component is defined both by the control state (that is, the state of the finite state machine) and the values of the local variables of the component.

As mentioned in Section 2.1, input and output patterns are used to describe the messages read from the input ports and written to the output ports. An input pattern is a pair of an input port name and a constructor pattern (in the functional style of Gofer) over the component's local variables and over free variables for the transition, separated by a "?". An input port pattern matches an actual message at this channel, if the constants and the values of the defined variables match the corresponding values of the read message. The free variables are bound to the actual values as found in the message. Thus, while the component's local variables are only read in an input pattern, the free variables get set during the matching process.

Output patterns are pairs of channel names and expressions, separated by a "!". The expressions have to fit to the type of the channel, and may be built over free and local variables.

In preconditions only defined (local component) variables may be used to formulate predicates over the current data state. In postconditions, defined and free variables can be used to define the predicates. Like in output patterns, defined and free variables are bound to their current values. Additionally, for each defined variable x a primed variable x' can be used to address the value of the variable after the execution of the transition.

It is important to note that in our approach the input is read simultaneously from all input channels. Formally, this influences only the semantics of the composition of automata, while we present here the semantics of single components, described by STDs.

Input patterns used in an STD are complete in the sense that unspecified combinations of input patterns are inter-

preted to result in an empty valued output and leave both control state and local variables unchanged. If no input pattern is defined for a certain port, the input pattern will only match if no message is received on this port. Output messages are treated analogously. If no output pattern is defined for a port, the value of the message written to this port is empty. As a consequence of this unbuffered semantics the meaning of STDs is closer to hardware oriented description mechanisms like Statecharts than abstract description languages like SDL (with input buffers at any component).

As mentioned above, transitions between the states S_1 and S_2 may be of the form:

$$S_1 \xrightarrow{P(x):I(x,v):O(x,v):C(x,x',v)} S_2$$

where

- x is the set of local variables defined in the DTD.
- v is a set of free variables, local to the definition of the transition.
- $P(x)$, the precondition, is a predicate over the variables of the state transition diagram x.
- $I(x,v) = I_1(x,v);\ldots;I_m(x,v)$ is a list of input patterns where the pattern $I_i(x,v)$ for port i_i may either be empty or $i_i?c_i(x,v)$ with $c_i(x,v)$ being a constructor pattern using the variables from x and v
- $O(x,v) = O_1(x,v);\ldots;O_n(x,v)$ is a list of output expression with the form $o_j!e_j(x,v)$.
- $C(x,x',v)$, the postcondition, is a predicate similar to the pre-condition, additionally using the primed variables x'.

For each transition of the above form a clause

$$\begin{array}{l}\exists v_1,\ldots,v_k.\\ \quad s = S_1 \wedge P(x) \wedge\\ \quad i_1 = c_1(x,v) \wedge \ldots \wedge i_m = c_m(x,v) \wedge\\ \quad t = S_2 \wedge\\ \quad o_1 = e_1(x,v) \wedge \ldots \wedge o_n = e_n(x,v) \wedge\\ \quad C(x,x',v)\end{array}$$

to describe the semantics of the component is introduced. The union over all states s and t of these clauses describes the transition relation of the automaton. If an empty pattern was used for channel i_i or o_j, $i_i = nil$ and $o_j = nil$, respectively, are used in the above clause. The complete transition relation is defined as the disjunction of all the clauses introduced for the defined transitions.

2.6. Consistency

If a specification exceeds the toy world size, the contained information in general is spread across several views, like in different modules or libraries of a large programming package. Many possibilities for errors and inconsitencies arise out of this fact. Here, simple syntax checks can already be an enormous help. Since AUTOFOCUS uses different classes of views as well as hierarchically organized view structures, it becomes even more important to make sure that the information spread out over several views is well-defined or "consistent" in a methodological sense. Therefore, consistency checks are offered to ensure that the produced views fit together.

The ports introduced as links between different abstraction levels are an important concept to formulate consistency criteria, since one port can be accessed by an inner and an outer view, which have to be consistent. Consistency includes several different classes of syntactical correctness criteria like:

Grammatical Correctness: The corresponding view obeys the syntactical rules for textual views or the graph-grammatical rules for graphical views

Views Interface Correctness: If a view is embedded into another view according to the hierarchical concepts introduced before, those views must have compatible interfaces to each other (components in SSDs, states in STDs, or boxes in EETs).

Definedness: If a view makes use of objects not defined in the view itself, those objects must be defined in a corresponding view (Channel types in SSDs or STDs, e.g.).

Type Correctness: The type of an object assigned and the type of the object it is assigned to must coincide (channels and ports in SSDs, or channel values and channel types in STDs, e.g.).

Completeness: All "necessary" views of a project are present (for simulation, e.g., each component must either have a defined automaton or a subsystem).

One form of consistency basically corresponds to the static analysis performed during the parsing of program code. In general, the grammatical correctness and the type correctness are checked here. Simple consistency conditions for a SSD are:

- Each component has a nonempty name
- Each connector is bound to a channel
- Each channel is bound to one port per direction with the same type

These conditions can easily be formalized using typed first order predicate logic with equality if we introduce appropriate individuals for the elementary objects like identifiers, components, ports, connectors, or directions, and appropriate functions like `name_of`, `type_of`, or `direction_of`. Thus, the last condition may be formalized as

$$\begin{aligned}&\forall chan : Channel.\\ &\ \exists icon : Connector.\\ &\ \ \exists ocon : Connector.\\ &\ \ \ \exists type : Type.\\ &\ \ \ \ \texttt{channel\_of}(icon) = chan \wedge\\ &\ \ \ \ \texttt{channel\_of}(ocon) = chan \wedge\\ &\ \ \ \ \texttt{has\_type}(icon, type) \wedge\\ &\ \ \ \ \texttt{has\_type}(ocon, type) \wedge\\ &\ \ \ \ \texttt{has\_type}(chan, type)\end{aligned}$$

Furthermore, we use checks that are in a similar way performed during the linking process of a program code. Primary checks for this case are the view interface correctness, the definedness and the completeness. Examples for conditions needing more than one view to be checked for SSDs are:

- For each port of a subview bound to the environment there exists a single port bound to the corresponding component of the superview having the same name, same type and opposite direction.
- If a component has a defined subview, then for each port of the component there exists a single port in this subview, having the same name, type and the opposite direction.

Again, these conditions can be formalized as above. Note that now, however, we have to add views as individuals and corresponding functions to the language.

2.6.1. Definition of Consistency Conditions

The actual conditions upon which consistency checks are based, are defined using a declarative textual notation, similar to first order predicate logic with a simple type system. This approach, in contrast to a more efficient hard-coding of consistency conditions, provides a clear and simple way for developers to extend the set of consistency conditions if needed. The definition of the consistency conditions also holds further information, like an informal explanation about the consistency condition and hints on how to overcome inconsistencies as supplied by the designer of the condition. The basic language elements of the textual notation used for the consistency conditions are

- quantors (universal quantor and existential quantor),
- selectors to access properties of specification, and
- operators on logical expressions and the equational operator.

An example for the completeness of a specification concerning simulation is given below in the concrete syntax used in the tool.

```
forAll c: Component .
        (exists ssd: SSDView .
                refinement(c, ssd)) OR
        (exists std: STDView .
                behavior(c, std))
```

This consistency condition relating components with SSD views and STD views asserts that each component must either be refined by an SSD view specifying its sub-structure or be related to an STD which specifies its behavior.

2.6.2. Integration of Inconsistency Treatment

In AUTOFOCUS, developers invoke the consistency checks from the project browser, the window for managing different developments projects, versions, and views. Since, as noted above, the developer may be interested in violating some consistency conditions, the developer is presented with a list of all consistency conditions; in case only a selection of the checks should be carried out, individual ones can be deselected in this list. The tool then performs the selected checks, and after having finished, presents the developer with a list of inconsistencies found including the views violating the consistency condition. Figure 1 shows the consistency checker interface of AUTOFOCUS.

From that list, it is possible to directly open an editor for each of the listed views where the components violating consistency conditions are highlighted. For consistency checks invoked from within editors, this is obviously not necessary, as the editor is already open; here the components violating consistency are directly highlighted.

3. Prototyping and Simulation

Prototyping and simulation can be an enormous help to developers. Using visualization techniques close to the description techniques used in the design of the system, properties of a system under development, for instance, behavior under specific environmental stimuli, can be conveniently observed and validated. Even in conjunction with formal development techniques prototyping can be used for the first iterations in a “round trip”-style engineering process to shape a first stable and basically reliable system design which is then further enhanced using formal verification techniques like model checking and theorem proving.

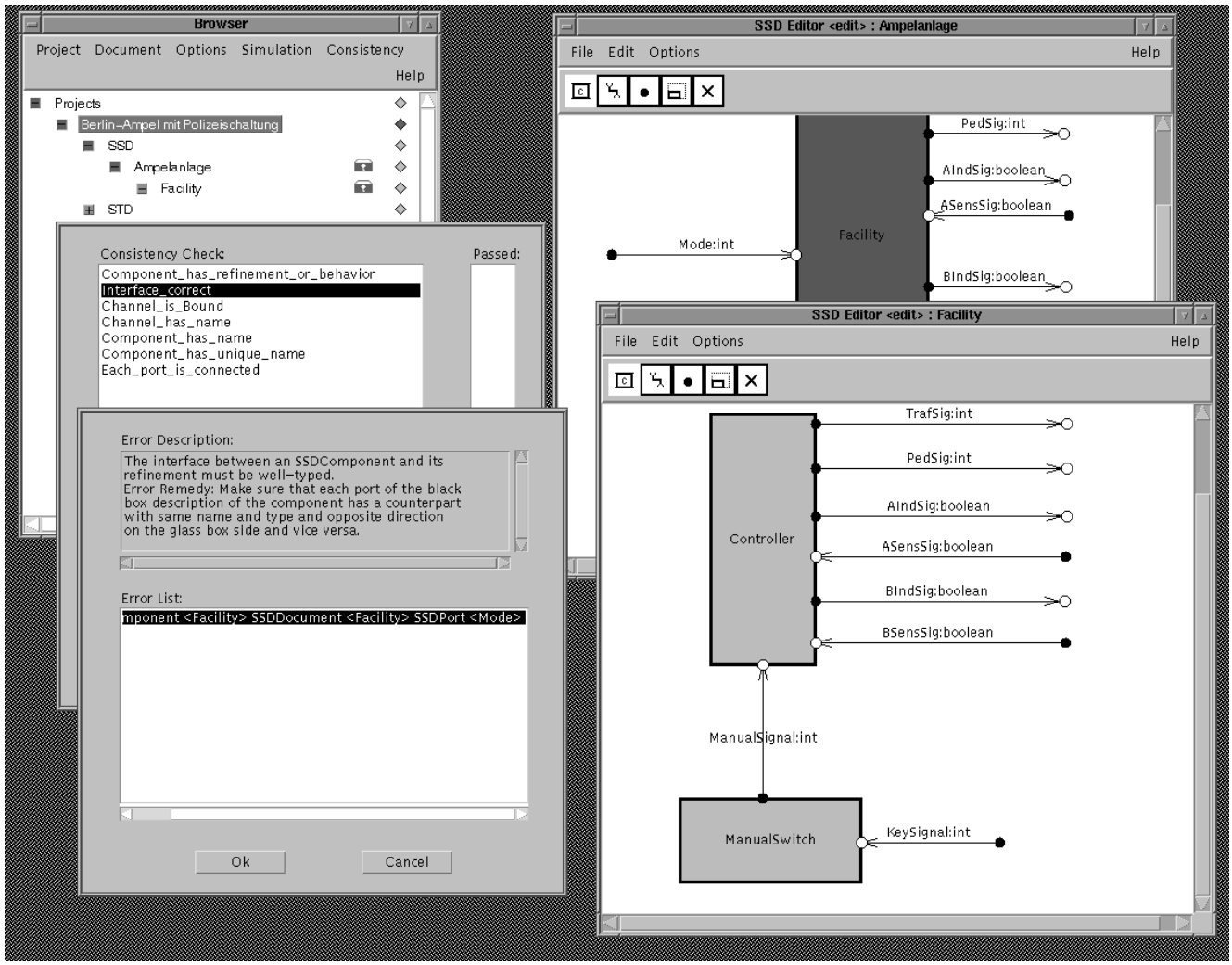

Figure 1. Treatment of Inconsistencies

Supporting this approach, AUTOFOCUS provides a tool component called "SIMCENTER" which offers facilities to

- generate executable prototypes of systems or parts thereof,
- run these prototypes in a simulation environment,
- visualize the runs using the same description techniques as used for designing the system, and
- optionally connect third-party front-ends like multimedia visualization tools or external hardware systems to the simulation environment.

3.1. Code Generation

As described in [8] we adopted a prototyping scheme based on generation of program code. As the language used for this purpose we chose the Java programming language for a number of reasons. First, AUTOFOCUS, including SIMCENTER, is entirely written in Java. Thus it is fairly easy to dynamically load the generated and compiled prototype code into the simulation environment using Java's Class Loader. Second, Java offers a promising future as implementation language for embedded systems applications. Having a working prototype generator for Java available would then simplify the implementation of code generators for "real" embedded applications.

The whole process of code generation can obviously only produce usable results if the system specification is consistent in the sense outlined in Section 2.6.

The code generation is based on the synchronous semantics outlined in Section 2 and in [8]. In the latter, where we describe the concepts of our prototyping approach, we suggested generating a Java class implementing the Java In-

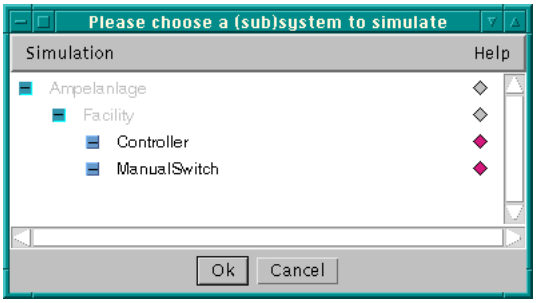

Figure 2. The Chooser Window for Selecting the Components to be simulated

terface *Runnable* for each component which would then be executed in its own, concurrent thread within the simulation environment. Enforcement of synchronicity required by the semantics chosen would, in this case, be guaranteed by the communication mechanism implemented by the communication channels. For reasons mainly of simplicity and manageability we chose a more "conservative" approach in the real implementation, which uses a central scheduler to control the progress of the simulation. This scheduler ensures that, within each clock cycle of the system, each component is enabled to perform one transition based on the input available and to produce its output.

Flexibility in Code Generation Obviously, in most cases it is not desirable to simulate a complete system under development, but only selected components thereof. In addition to this choice AUTOFOCUS allows the developers to select the granularity of abstraction, for the simulation by picking just the system parts they are interested in. AUTOFOCUS uses a hierarchical chooser window. The contents of this chooser window exactly represent the component hierarchy as specified in the SSDs (see Fig. 2).
However the user can only choose the simulation between the STD of a component and the simulation of the subcomponents, if both are available. If there is no STD for the component, or if there is no SSD with "simulateable" subcomponents, there is no possibility to choose. The chooser window checks this and allows only correct choices.

Properties of the Generated Code The hierarchical component structure used for simulation is always a tree with components at its leafs where STDs have been assigned to. In other words, the behavior relevant for the simulation of the system or chosen parts can be found at the leaf components. All other components do not contribute to the system's behavior but, instead, act as structuring mechanism during development.

For this reason, the generated Java code does not include structural information and is, therefore, "flat". For each leaf component a corresponding Java class is generated and connected to other classes accordingly.

3.2. Interactive Visualization of Code Execution

Executing a generated prototype in a simulation is not very helpful if the simulation cannot adequately be visualized to developers. Usually, debugging and visualization of code execution are based on the program code. But the generated flat code (see Section 3.1) does not properly reflect the hierarchical properties of the specifications. Moreover, program code is at a very high level of detail containing much information irrelevant for validating specifications.

The appropriate level of visualization and debugging is, in our view, the same level at which developers design their specifications: Different kinds of graphical description techniques with a hierarchical refinement relationship (see Section 2). Consequently, SIMCENTER provides so-called *Animators* for each description technique. Animators provide the same look-and-feel as the corresponding editors, but they do not allow editing. Instead, animators just visualize the animated specification and highlight the currently active parts. Furthermore, animators allow the user to monitor and even modify the state of the program.

SSD Animators visualize the structure of a system during simulation. Channels on which messages are sent are highlighted, and the values of the messages passed along them are attached to them.

STD Animators show the states and state transitions performed by components during a simulation. Current states and firing transitions are highlighted. Users can change the active state.

Variable Animators display the values of state variables of components during a simulation and allow users to change these values.

EET Animators show and record the communication history of selected components in a simulation. These communication histories provide a graphical runtime protocol of a simulation.

Figure 3 shows these animators during a simulation run of a pedestrian traffic lights system example.

As the generated code is flat (see Section 3.1), but the corresponding animators are eventually hierarchical, more than one animator stands for a specific part of the code. For example, if a component is refined by a network of subcomponents and one of these subcomponents is active the corresponding supercomponent has to be visualized as active as well. Thus, animators *observe* the *observable* program code and will be notified in case of relevant changes of the system state.

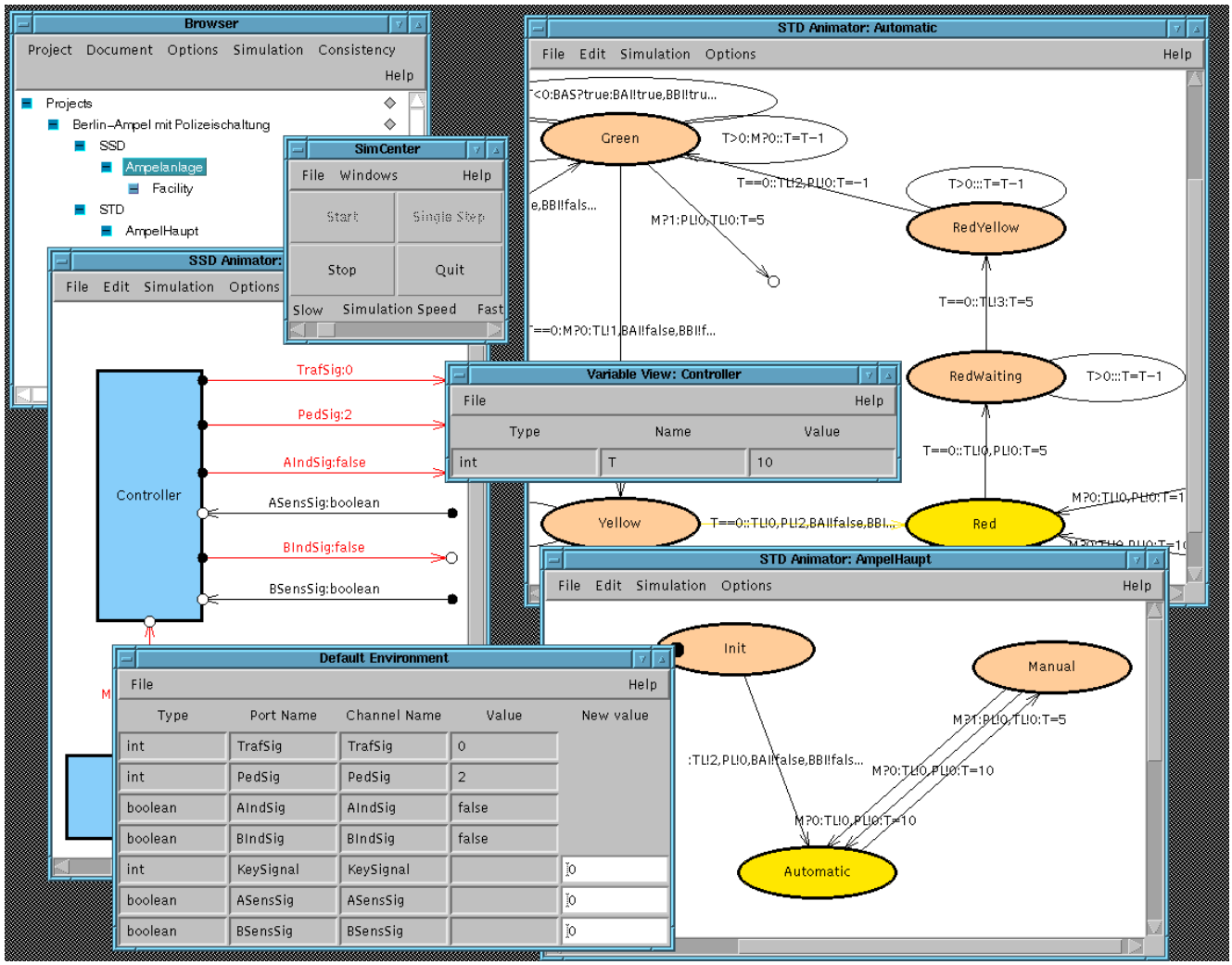

Figure 3. A running simulation in SimCenter showing different kinds of animators and viewers

3.3. Providing an Environment for Simulation

Especially when developing embedded systems software it is desirable for developers to gain a more application-oriented animation of a system's properties than available by the simulation animators introduced in Section 3.2. In this context, we understand the notion "application-oriented animation" as an animation oriented towards the environment into which the software will be embedded and towards the user interface behavior potential users of the system (customers) will experience.

Using the Default Environment An environment view shows the values present on all channels that connect the system to its environment. Output channels are displayed read-only, whereas the contents of input channels can be modified by the user in order to feed the system with external stimuli. In the left bottom corner of Figure 3 a standard environment is shown. With inputs in the white fields labeled "New value", the user can modify values on the input channels of a system.

Connecting to Third-Party Systems To further enhance the visualization of simulation runs, SIMCENTER allows to connect a user-defined front-end to the simulation environment using a standardized interface and communication mechanism. Communication between SIMCENTER and the font-end works both ways: the state of the simulation can be visualized in the front-end, but the front-end can as well generate, for instance by user interaction, inputs for the system being simulated. Thus, the simulated system is not only

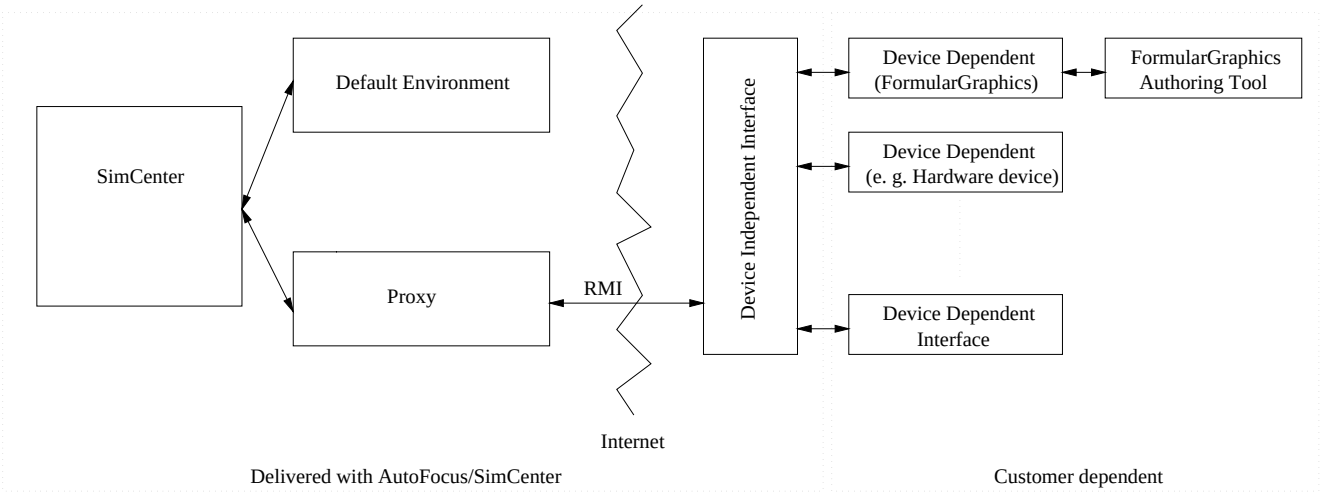

Figure 5. The SimCenter interface to third-party systems

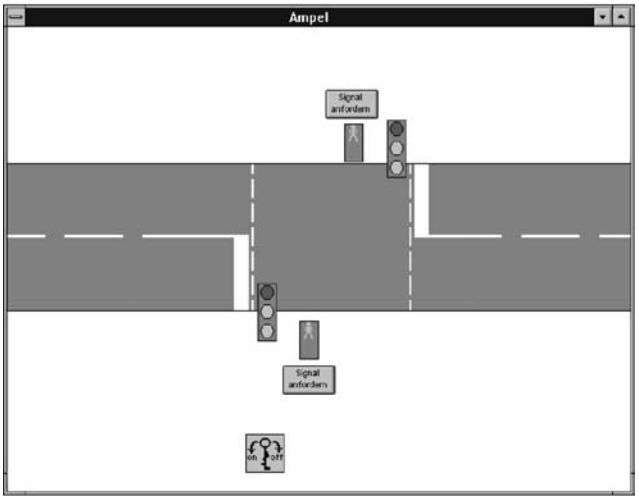

Figure 4. A visualization of a simulation running in SimCenter using the third-party front-end system Formula Graphics

observable in its environment, but it also *observes* its environment, just as a reactive embedded system is supposed to.

In the case of our pedestrian traffic lights controller system, a visualization of the whole system at its user interface level, that is, at the level of the traffic lights, is desirable. Using a standard multimedia authoring tool, *Formula Graphics*, and some glue program code to attach it to SIMCENTER's communication interface, the user-defined visualization shown in Figure 4 can be attached to the simulation environment. It can then be used to display the progress of the simulation as well as to obtain input from a user of the system, for instance a request for "green" for the pedestrians.

The connection between SIMCENTER and the simulation environment is realized using the Remote Method Invocation (RMI) package of Java. The advantage of this solution is that SIMCENTER and the simulation environment can run on two different platforms and/or operating systems. The two systems only need to be connected via the Internet. Thus it is possible to execute SIMCENTER on a UNIX workstation and, for instance, a multimedia authoring tool like *Formula Graphics* on a Windows system or a hardware control on a real time computer. That means that developers are able to choose the appropriate platforms and are not bound to a specific one.

Through the fact that the design of the RMI interface, shown in Figure 5 is generic, users only have to adapt the device dependent interface, which is a straightforward task. So it is even feasible to attach an embedded system to SIMCENTER using the same interface mechanism.

4. Conclusion and Future Work

FOCUS is a formal method for the development of distributed systems, interacting with their environment. Based on these concepts, AUTOFOCUS supports graphical development of such systems.

Future activities will bring AUTOFOCUS closer to its formal basis. This will increase the flexibility of software engineering and abstraction, since mathematics are more expressive than graphics.

The planed extensions of AUTOFOCUS are:

- connection to a verification tool to prove refinements between specifications,
- connection to a model checker, to verify finite problems automatically, and
- the integration of test methods to validate conventionally developed code.

Acknowledgments

For the work on AUTOFOCUS we thank all active students in our software engineering courses, especially G. Einert and G. Klein. For some comments we thank our anonymous referees.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] G. Booch, I. Jacobson, and J. Rumbaugh. *UML Summary*. Rational Software Cooperation, Jan. 1997. Version 1.0.

[3] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. Gritzner, and R. Weber. The Design of Distributed Systems - an Introduction to FOCUS. Technical Report TUM-I9203, Technische Universität München, Institut für Infor matik, Januar 1992.

[4] M. Broy and S. Jähnichen, editors. *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, volume 1009 of *LNCS*. Springer, Heidelberg, Nov 1995.

[5] R. Grosu, C. Klein, B. Rumpe, and M. Broy. State Transition Diagrams. Technical Report TUM-I9630, Technische Univerität München, 1996.

[6] D. Harel. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.

[7] F. Huber, S. Molterer, B. Schätz, O. Slotosch, and A. Vilbig. Traffic Lights - An AUTOFOCUS Case Study. In *International Conference on Application of Concurrency to System Design (CSD'98)*, 1998. To appear.

[8] F. Huber and B. 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.

[9] F. Huber, B. Schätz, and G. Einert. Consistent Graphical Specification of Distributed Systems. In P. L. John Fitzgerald, Cliff B. Jones, editor, *FME '97: 4th International Symposium of Formal Methods Europe, Lecture Notes in Computer Science 1313, pp. 122 - 141*. Springer, 1997.

[10] M. P. Jones. *An Introduction to Gofer*, Aug. 1993.

[11] B. Schätz, H. Hußmann, and M. Broy. Graphical Development of Consistent System Specifications. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances In Formal Methods*. Springer, 1996.

[12] Telelogic AB. *Telelogic AB: SDT 3.1 Reference Manual*, 1996.

[13] Z.120. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-TS, Geneva, 1996.