T U M

I N S T I T U T F Ü R I N F O R M A T I K

Integrated Development of Embedded Systems with AutoFocus

Franz Huber, Bernhard Schätz

TUM-I0107
Dezember 01

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-12-I0107-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Integrated Development of Embedded Systems with AutoFOCUS*

Franz Huber, Bernhard Schätz
Institut für Informatik, Technische Universität München,
Arcisstr. 21, D-80333 München, Germany
{huberf|schaetz}@in.tum.de

December 4, 2001

Abstract

This article presents AutoFOCUS, a tool prototype for formally based development of distributed, embedded systems. AutoFOCUS supports system development offering integrated, comprehensive and mainly graphical description techniques to specify different views as well as different levels of abstraction of a system. To avoid ill-defined specifications, consistency conditions on these system descriptions can be formulated and checked. Prototypes can be generated from consistent and executable specifications. These prototypes can be executed and visualized within a simulation environment. For formal verification of specification properties, common model checkers can be used with AutoFOCUS.

1 Introduction

During the last few years the use of embedded systems has massively grown. Increasingly, functionality formerly implemented by hardware is replaced by more and more complex software components, often with new functionality added. The development in the automotive industry is a good example for this tendency: formerly isolated, comparably simple systems such as anti-skid brake systems, engine or cruise control systems have evolved into networks forming global automotive control systems. This rapidly increasing functionality and complexity of embedded software requires the use of software engineering techniques and supporting tools in embedded systems development, while its safety-relevance requires the application of rigid or even formal techniques.

1.1 Motivation for AutoFOCUS Development

Since tools for development of distributed, embedded systems are already commercially available, the question arises why there is need for yet another academic tool prototype. There are several answers to this question.

First and most important, even though there are several commercial CASE tools for embedded systems around, most of those tools either lack sufficient sound conceptual foundation while concentrating on more practical aspects like code-generation and convenient user interfaces, or offer a well-founded conceptual and mathematical basis but are too mathematically oriented for the systems engineer (see Section 1.2 for a more detailed discussion). Thus, the combination of usability and simple but precise concepts of modeling still remains a rewarding area of research.

Furthermore, besides developing a tool concept supporting core aspects of the FOCUS development methodology for distributed systems [8], as a side effect, important knowledge in questions of CASE tool

*The authors of this paper were funded by the DFG-Sonderforschungsbereich 342 “Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen” and the project “SYSLAB” supported by DFG under the Leibniz Program and Siemens-Nixdorf.

design, tool usability and management issues of large[1] software development projects could be gained. Additionally, using the AutoFOCUS prototype a platform for the evaluation of different methodological concepts was established supporting experiments with different development strategies.

1.2 Related Developments

The spectrum of currently available tools ranges from ones for mainly verification oriented approaches to those focusing on simulation and code-generation, like [1] or [12] and [27]. In general, however, these approaches only to a limited extent try to integrate the different aspects of a formal development process, create an integrated combination of intuitive and engineering-oriented description techniques, a strong semantical basis and verification support as well as validation by simulation and code generation.

For example, UML [5] includes several description techniques and offers tool support, such as Rational Rose ([24]), for the specification of systems using those techniques. However, UML lacks a precise semantical base and therefore neither allows sophisticated consistency checks between different parts of a specification nor supports verification of system properties. Similar observations also hold for tools using formally well-founded description techniques, like SDT ([27]) or Rhapsody ([18]). On the other hand, more formally oriented tools like STeP ([4]) or Atelier B ([1]) often lack methodical features like integrated views or the ability to create and relate system description using different levels of abstraction.

The aim for developing AutoFOCUS was not to build a competitor to those commercially available products. The major focus was to investigate, how those experiences gained in prior fundamental research projects could be applied consequently for the development of software for embedded systems. Central aspects of consideration are: semantically sound description techniques, the modularity of those description techniques to structure specifications using integrated views on different levels of abstractions, a methodical application of those techniques for an integrated modeling and development of embedded software. From this point of view, approaches like StateMate are not rigid enough considering the separation of views, for example, by mixing structure and behavior (using both AND- and OR-states in one diagram). Furthermore, the application of the possibilities of the formal basis by adding proof tools like model checkers or theorem provers is lacking. While other approaches support this separation of views, the use of completely specified description techniques is not applied. Thus, for example, in the ObjecTime approach it is necessary to enrich the automata diagrams with program code for a complete behavioral description. Furthermore, ROOM ([26]) and ObjecTime were described without defining a formal model; the semantic properties of its systems are only defined by the generation of executable models and thus on a very implementation-oriented level. Thus, besides possibly introducing open questions concerning the modeling concepts, the necessary requirements to connect formal verification tools is lacking.

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.

2 Methodological Background

Since a tool is as only as good as its methodology, we first have to lay down a methodological basis to build a tool upon. Therefore we need to define

- what building blocks are needed to build a specification,
- how those building blocks are presented to the user,
- how a specification is built using those representations, and
- how parts of specifications can be reused in another specification.

Each of these questions will be discussed in turn by the following subsections.

[1] "Large" in this context has to be seen with respect to project sizes of typical university programming projects.

2.1 Modeling Concepts for Distributed Systems

Independent of how we represent a specification of an embedded system, we can identify some basic concepts characterizing such systems. Subsequently, we briefly describe the conceptual elements that make up our view of a distributed system.

Components are units encapsulating data, structure, and behavior, communicating with their environment. They are reactive and respond to stimuli from the environment. Components can be hierarchically structured, i.e., consist of a set of communicating sub-components.

Data types define data structures used by components. Data types are described in terms of product and summation types.

Data items are encapsulated by a component and provide a means to store persistent state information inside a component. They are realized by typed state variables.

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

Channels are directed, named, and typed. They connect component ports. They define the communication structure, i.e., the topology, of a distributed system.

Control States and Transitions between their entry and exit points—called Connectors—define the flow of control within a component.

Behavior describes how a component reacts to input from its environment depending on the state of the data encapsulated by the component and the state of control, given by the component's internal state-based control flow description.

With respect to the underlying mathematical formalisms of FOCUS, the elements in this conceptual model can be regarded as abstractions of both the formal model and the concrete notations used to describe them. Thus, the conceptual model represents the common denominator of both the description techniques and a formal model.

2.2 Views and Notations: Description Techniques

To form a comprehensive and structured picture of a system, it should be described from different points of view and on different levels of abstraction. Therefore, AutoFOCUS offers four hierarchically structured description techniques:

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

each one covering different—yet not necessarily disjoint—aspects the system. The integration of the views on a common semantic basis leads to one integrated formal system specification of the system.

AutoFOCUS supports hierarchical development of systems. Depending on the granularity, components or views can be atomic, or consist of sub-components or sub-views themselves. Therefore, AutoFOCUS allows to switch between different levels of granularity by using the hierarchical description techniques described in the following sections.

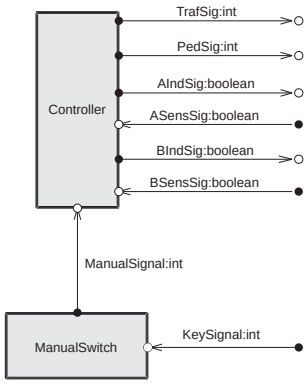

Figure 1: A Sample System Structure Diagram

2.2.1 System Structure Diagrams (SSDs)

A (distributed) system consists of its components and the communication channels between them. An embedded system communicates with its environment. To describe the static aspects of distributed systems, viewing it as a network of interconnected components exchanging messages over channels, we use system structure diagrams (SSDs). Graphically, SSDs, as shown in Figure 1[2], are similar to data flow diagrams, represented by graphs with labelled rectangular nodes symbolizing components, arrow-shaped labelled edges symbolizing channels, and circles at both ends symbolizing ports. Each component has a name and a set of input and output channels attached to it via input and output ports. Every channel is defined by a channel name and the set of messages that may be sent on it. In case no value is sent, the channel contains a default nil-value. Thus SSDs provide both the topological view of a distributed system and the signature of each individual component.

Since each component can be described as distributed system in itself by assigning an SSD to it, hierarchical system descriptions can be specified. Here, ports are used for modular descriptions: a port attached to a component will also be present in the inside view of this component, i.e., the assigned SSD. Thus visible from the inside and the outside, ports serve as interface between the environment of a component and its internal structure.

In AutoFOCUS components in an SSD can be associated with substructures (SSDs), behavioral views (STDs or EETs), and component data declarations (CDDs). A component data declaration declares local variables for the component by setting a name and a data type for each variable. These variables are used to define a components' behavior by characterizing the data space of the component.

2.2.2 Datatype Definitions (DTDs)

In AutoFOCUS, the types of the data processed by a distributed system are defined using textual notations called data type definitions (DTDs). Currently, two different approaches to deal with data in system specifications are provided.

Java: A subset of the data types available in Java can be used to define data elements. Creation of user-defined data types is not possible here.

Functional: The basic types and data type constructors from FriscoF [22], a functional programming language derived from Gofer [20], can be used to declare data elements and to define new data types.

Both approaches to data and data type definition are integrated into the prototype generator, thus projects with data definitions in both languages can be simulated.

Data types (pre-defined basic types or user-defined ones) usually are used to declare local component variables (see above), local transition variables or to define the types of communication channels and ports.

[2] The diagrams and screenshots in this paper are taken from a case study done with AutoFOCUS, a pedestrian traffic lights controller system.

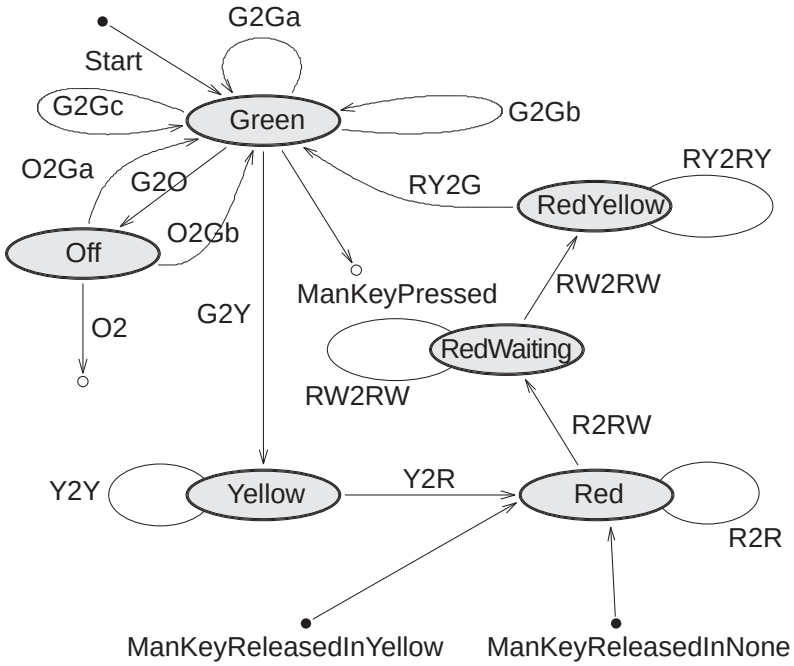

Figure 2: A Sample State Transition Diagram

2.2.3 State Transition Diagrams (STDs)

An STD characterizes the behavior of a system or component reacting to input received from its environment and producing output sent to the environment depending on the actual state of the component and setting a new state. Because 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 (the state of the finite state machine) and the values of the local variables of the component.

STDs are extended finite automata similar to those introduced in [11]. Graphically, STDs are represented by graphs with labelled oval nodes as states and labelled arrows as transitions. Figure 2 shows a simple example. Each system component of an SSD can be associated with an STD. Just as with SSDs, each state of an STD can be decomposed into an STD of its own. Transitions are characterized by the following annotations:

- pre- and post-conditions, which are basically predicates over the local data state of the component as described by its local data definition,
- a set of input and output patterns describing the messages that are read from or written to the input and output ports of the component, and
- an optional label, to complement the otherwise used conditions and patterns for better readability.

For hierarchical decomposition of states in STDs, we use a concept similar to the one used in the SSDs. The treatment of substructures in STDs and SSDs is similar: for every edge (transition/channel) into or from the node (component/state) a connector or port is created both at the node and in the substructure. They can be used to formulate syntactic consistency conditions, as described in Section 4.

As previously mentioned, 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 over the component's local variables and over free variables for the transition. An input port pattern matches an actual message at this port, 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

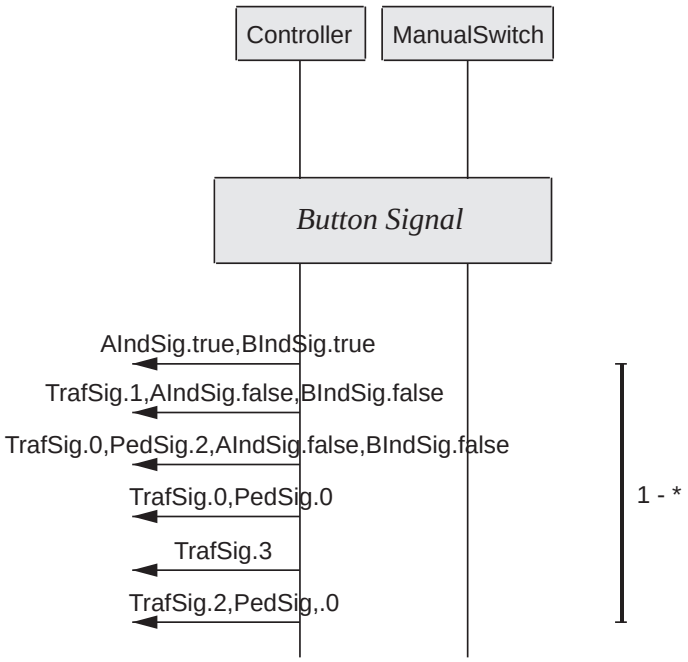

Figure 3: A Sample Extended Event Trace Diagram

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 port names and expressions. In output patterns, defined and free variables are bound to their current values. In post-conditions, 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 ports. 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 interpreted to result in an empty-valued output and leave both control state and local variables unchanged.

2.2.4 Extended Event Traces (EETs)

Aside from STDs, extended event traces may also be used to describe the behavior of distributed systems. EETs (see Figure 3) assume a component- and communication-oriented view, describing system behavior by sample communication histories between components. AutoFOCUS uses a subset of notational elements of the MSC standard [19], similar to the concepts used in UML ([5]). As well as other graphical AutoFOCUS notations, EETs support hierarchical concepts. Boxes can used to reference a set of sub-EETs. This means of structuring also allows the introduction of variants of behavior. Additionally, indicators can be used to define optional or repeatable subparts of an EET. A complete description of EETs can be found in [25]. EETs can be used at different development stages for 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.

2.3 An Outline of the Development Process

System development in AutoFOCUS is basically a top-down process. As depicted in Figure 4 a system is initially specified as a black box component interacting with its environment. Interaction is accomplished

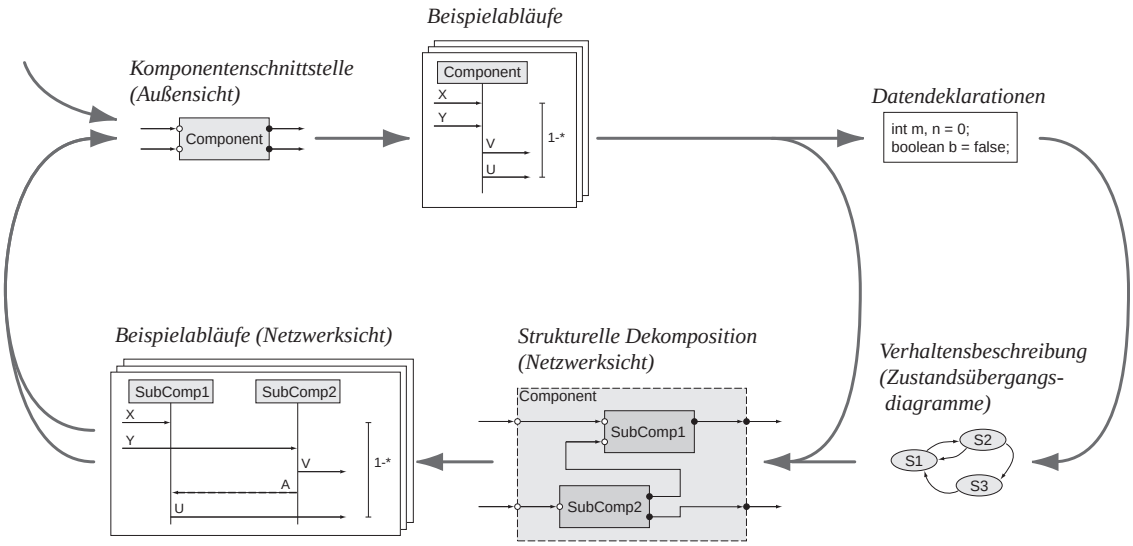

Figure 4: An Outline of the AutoFOCUS Development Process

using the component's interface given by its set of ports. Basic functionality of the system can be specified at this point in development using sample communication runs between the system and its environment. These communication histories are given by EETs. It is possible to specify a rudimentary behavior for the complete system by assigning an explicit control flow specification given as a state transition diagram to the system component as a whole. If needed, data stored in the component can be declared.

Usually, the system will first be decomposed into a set of subcomponents, each of which is assigned some more specific functionality than to the system as a whole. To interact with each other, the subcomponents form a network connected by channels. This process of decomposition can be repeated as many times as necessary, resulting in a hierarchical system structure that becomes more and more fine grained with each iteration.

With the distribution of system functionality to different components in the system, the sample runs specified for the upper-level components have to be projected and distributed to these components (see lower part of Figure 4). Again, for each component in the decomposition hierarchy a collection of local, encapsulated data elements can be declared. Likewise, a control flow specification can be given by state transition diagrams for each component (see right part of Figure 4). For a complete case study, which demonstrates this development process, we refer to [13].

2.4 Reuse of Specifications and Components

Within the development process just outlined it is, of course, desirable that not all components specified during the decomposition process are created anew. Previously specified components, possibly from a library of predefined components should be reused and integrated into a new system. Such practice is an inherent concept of the component-based approach of AutoFOCUS; however, in the current implementation of the tool it is not yet possible in a straightforward manner. The main reason for this is a technical one: AutoFOCUS currently still uses a document-based repository (see the following section). This makes it difficult to realize a model element-based[3] approach to reuse, whereas a reuse approach on the basis of documents is, of course, straightforward. As pointed out in Section 9 we are currently working on a model-based repository to replace the current document-oriented version.

[3]Here, components are elements of an abstract "syntactical" model. They contain more than is visible in a structure diagram document, such as their data elements and their control states, both of which are given in separate diagrams (documents). In such a document-oriented approach, reuse becomes a matter of "cutting out" information form several documents.

3 Basic AutoFOCUS Features and Architecture

3.1 Client/Server Architecture

Because of their complexity distributed systems are often developed in teams by several developers at the same time, often using different computer platforms. Therefore, AutoFOCUS is implemented as a client/server system with a central repository where all development documents are stored. An arbitrary number of clients can access these documents over a network connection. Thus system developers can use the specification documents simultaneously. By implementing the clients in Java, AutoFOCUS can be used on most of the common operating system platforms.

Specifications are repeatedly revised, especially in the early phases of development. Therefore, the possibility to use version control of single documents as well as of whole projects is absolutely necessary. Tool support should allow to rule out inconsistencies caused by team members working on the same specification documents. The AutoFOCUS repository offers version control of both documents and projects as well as locking mechanisms for documents based on the usual pattern of single write access and multiple read accesses.

3.2 Specification Tools

On the client side, the complete functionality of AutoFOCUS is accessible in a graphical user interface, parts of which are shown in Figure 5. A project browser provides access to the development projects in the repository as well as to the associated document hierarchies grouped by document classes. For each of the graphical description techniques introduced in Section 2.2 AutoFOCUS provides a (graphical) editor. These editors, supporting the hierarchy concepts of FOCUS, are also shown in Figure 5. All editors use an identical user interface concept with mouse-based interaction, clipboard functionality, basic graphical alignment support, and a basic form of automatic routing of channels/transitions to facilitate editing development documents. To make AutoFOCUS diagrams available for documentation purposes these diagrams can be exported into encapsulated PostScript graphics files and thus used in common word processors.

Document-oriented description In AutoFOCUS, a project, representing a system under development, consists of a number of documents that are representations of views using the description techniques introduced above. Thus each description technique is mapped to a corresponding class of documents (also called diagrams). Combined, these documents provide a complete characterization of a system in its current development status. Access and version control is done on the document level of granularity in the repository, which keeps track of the complete version history of every document. In order to reuse documents, a document may be referenced by more than one project. Projects are subject to version control as well: in our approach, versions of a project are collections of specific versions of development documents.

Hierarchical documents As mentioned before, all graphical AutoFOCUS description techniques share the concept of hierarchy. Both system structure diagrams and state transition diagrams—which are essentially graphs—as well as extended event traces allow hierarchical decomposition. The glass-box view of a component in a system structure diagram is specified in a second structure diagram document. In the same way, a state in a state transition diagram can be characterized by another state transition diagram document describing this state on a more detailed level. In extended event trace diagrams, so-called ”boxes” (see Section 2.2.4) are introduced as an abbreviating notation for parts of system runs specified in different event trace diagrams.

Integrated views From the user’s point of view, the documents of a development project are tightly integrated, both vertically along the refinement hierarchies and horizontally along the relationships between documents of different kinds. These relationships between the different document kinds are in turn based on the relationships between the modeling concepts contained in them, i.e., on the underlying conceptual model (see Subsection 2.1). For instance, a state transition diagram can be associated with a component in a structure diagram denoting that this state transition diagram specifies the behavior of the component. Along relationships like these, quick and intuitive navigation mechanisms between the documents are available.

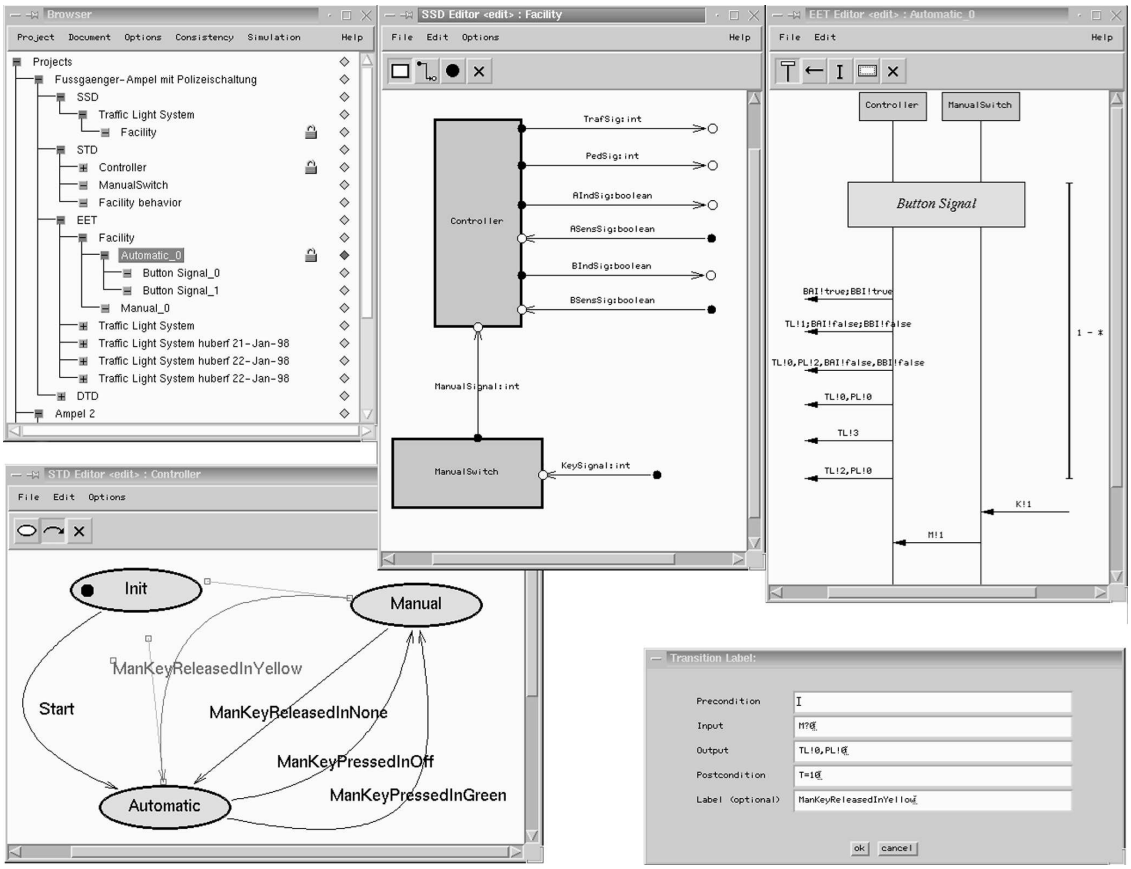

Figure 5: The AutoFOCUS Client Application

4 Consistency Control

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 inconsistencies 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 that have to be consistent.

4.1 The Notion of Consistency in AutoFOCUS

Consistency includes several different classes of syntactical correctness criteria like:

Grammatical correctness: The 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 STD 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 an SSD are:

- Each component has a nonempty name.
- Each port 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

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

Furthermore, we use checks that are performed in a similar way 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.

4.2 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 an easy way for developers to extend the set of consistency conditions if needed. The definition of the consistency conditions also contains 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 specifications, 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.

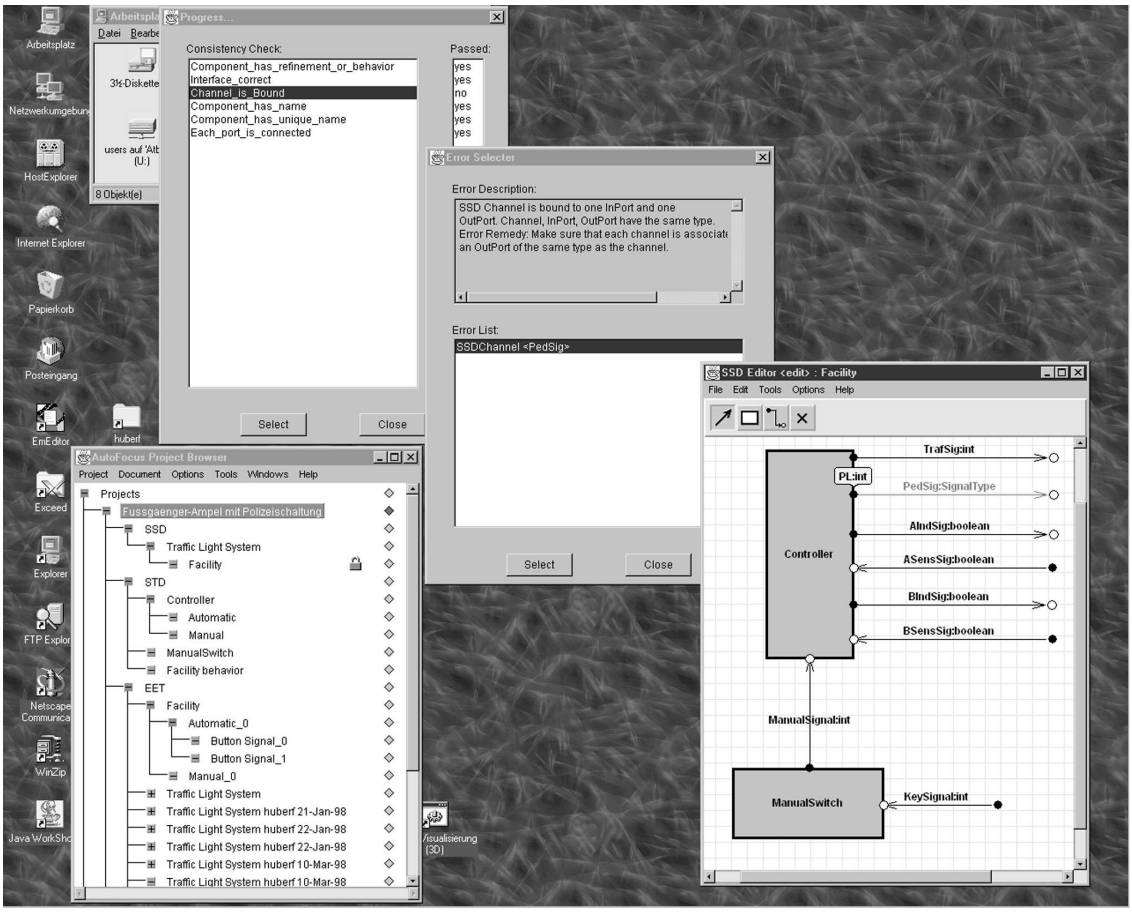

Figure 6: Treatment of Inconsistencies

```
forAll c: Component .
  (exists ssd: SSDView.
      decomposition(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 decomposed into subcomponents shown in an SSD view specifying its sub-structure or be related to an STD which specifies its behavior.

4.3 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, during a development process the global consistency of a project is not fulfilled most of the time developers are 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 6 shows the consistency checker interface of AutoFOCUS. From that list, it is possible to directly open an editor for each of the listed views with the components violating consistency conditions highlighted.

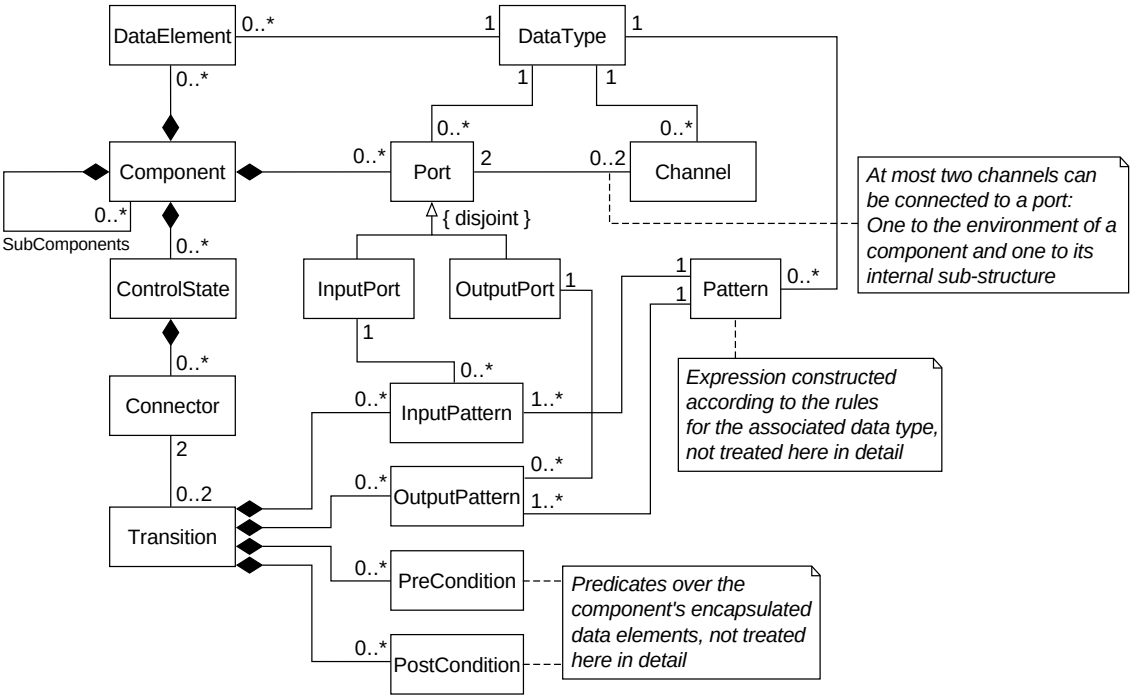

Figure 7: Simplified Conceptual Model

5 Consistency Conditions, Conceptual Model and Model Based Process

In the sections above, the notions of modeling concepts, of description techniques and of consistency conditions have been introduced. Based on these notions, we will know describe in more detail the notion of the *conceptual model* which lies at the core of the AutoFOCUSapproach and aims at supporting a formal development process.

5.1 Conceptual Model

This subsection introduces a simplified version of the conceptual model for describing distributed systems in our method. Its instances are abstract specifications describing systems. Developers create and manipulate them by means of concrete notations, which provide views upon them, possibly even several different graphical or textual views on the same parts of the model. Some of the notations used have been introduced in [17], and an example was given in Section 4.

From a description technique-oriented point of view, the elements in the conceptual model make up the essence (or the abstract syntax) of the information given by the notations used, quite similar to abstract syntax trees generated by parsers for common programming languages. In programming languages, however, source code documents are the important modeling concept and the syntax tree is just generated by the parser, in general unnoticeable from the users perspective. Many software engineering tools offer a similar approach, treating system descriptions as - at most loosely related - documents of different kinds. In our methodology, the abstract model is the central concept, both from the tool developers and from the methodologists point of view, and developers deal directly with the elements of the abstract model - without encapsulation in artificial constructs such as documents. The modeling elements of our example method are shown in Figure 2 in a UML-style notation. For a more detailed description of the modeling concepts we again refer to [17], a more detailed description of the conceptual model can be found in [7].

The most prominent elements and relations of the conceptual model are

- Components and their association to data types, local variables, ports, behavior and sub components

- Data types and their associated subtypes
- Ports and their associated channels and types
- Channels and their associated types and ports
- Control States, Transitions, Connectors, preconditions, postconditions, input and output patterns, and their connecting associations as well as associations to components, data types, local variables, ports substates.

With respect to an underlying formal model, the elements in the conceptual model can be regarded as abstractions of both the formal model and the concrete notations used to describe them, such, the conceptual model represents the common denominator of both the description techniques and a formal model. Elements from the conceptual model are represented both by notations used by developers and by formal (mathematical, logical) concepts used, e.g., to conduct proofs of system properties. Viewing specifications as graphs, as sketched in 2.2, it is obvious that, when using only the elements and relationships in the conceptual model (leaving aside the arities given for the relation-ships), it is possible to construct a multitude of such graphs. Then, of course, most of the possible graphs will not conform to the conceptual model . In this respect, the conceptual model acts as a requirement specification for well formed, discriminating well formed from ill formed specifications.

5.2 Consistency Conditions

As described, consistency conditions exist in three different levels:

Invariant conceptual consistency conditions: These conditions are expressible within the conceptual framework. They must hold invariantly throughout the development process. Therefore, they are enforced by construction - this is achieved by defining them on the level of the conceptual model.

Variant conceptual consistency conditions: Like the invariant conceptual conditions, these conditions can be expressed completely within the conceptual model. However, unlike those, they may be relaxed during certain steps of the development process and are enforced during others.

Semantic consistency conditions: These conditions are not - completely - expressible in the product model. Of course, from a theoretical point of view, it might be interesting to distinguish semantical conditions which must hold throughout the development process. Since, however, these conditions cannot be enforced using the conceptual framework, the validity of these conditions cannot be guaranteed throughout the development process but must be checked at defined steps of the process. Thus, from a methodical point of view there is no necessity to distinguish these forms of consistency conditions.

In general, defining a conceptual model requires to allow all three forms of consistency conditions. Examples for each from within the AutoFocus framework are:

Invariant conceptual consistency conditions: Syntactic consistency conditions like ”A channel connects two port”, ”A port is associated to a component”, ”A transition connects two states”.

Variant conceptual consistency conditions: Methodical consistency conditions like Port and associated channel are of the same type”, ”An event trace associated to a component only uses components which are subcomponents of the associated component”.

Semantic consistency conditions: Semantic conditions like ”The glass box behavior of a component refines the black box behavior”, ”The behavior of an event trace of a component is a refinement of the behavior of the component”.

While the distinction between semantic and conceptual consistency conditions is rather straight-forward, the distinction between invariant and variant conceptual consistency conditions is far less obvious. In general, the distinction between these conditions is a question of flexibility and rigorousness of the development process supported by the underlying model.

5.3 Views, Consistency and Process

The notion of a view as used here is a special application of the notion of a view as found in the 'model-view-controller' (MVC) approach. In the integrated model-based approach, each view of the system contains information specific to the view which is needed to define a complete model of the system. Thus, while in the MVC-approach a view is often interpreted as a mere form of graphical representation of the same model, here views differ essentially concerning the information contained. All views together form the model; additional graphical information (like the position of components, channels, states, etc.) is not considered to be part of the view but only of the presentation of the view.

We therefore define the *conceptual model* as the collection of all consistency conditions which must hold throughout the development process. As explained above, those conditions are restricted to conceptual consistency conditions.Thus the conceptual model can be interpreted as the collection of all elements of the views (abstracting from their graphical representation) and all relations which hold invariantly. All other consistency conditions (both conceptual and semantical) may be violated throughout the development process and must be checked if needed as precondition to certain development steps.

Thus, the definition of a conceptual model and the corresponding system development process are closely related. For instance, in a rigorous process channels might be only introduced between ports or must have a type assigned. In a more flexible process, it might be possible to create a channel starting from a port and leaving both end port and type unfixed at the time being. Therefore, for a rigorous process all conceptual consistency conditions are integrated into the conceptual model; while this limits the construction possibilities of the designer by abolishing inconsistent views, it avoids the need for checking variable consistency conditions at certain steps. On the other hand, reducing the set of invariant consistency conditions leads to a more flexible process at the price of possibly inconsistent views.

Increasing the rigidness of the conceptual model leads to what we call *model-based development process*. This differs essentially from a graphical-oriented development process, since the descriptions are not mere drawings but conceptually sound models. Most obviously, the limitations of the conceptual model lead to an increased quality of description of the system by eliminating unsound models. However, model-based development focuses on increasing the efficiency of the development process. The two major advantages of model-based developments are:

Abstraction: Using the conceptual model, we can abstract away from implementation details. Thus, for instance, reactive behavior can be modeled as changing states and producing output triggered by input instead of a programming a control loop consisting of labeled blocks consisting of assignments. At the same time, the increased abstraction also leads to *more structured descriptions* of a system. Instead of blocks of assignments, preconditions, postconditions, analysation of input, and generation of output can be distinguished.

Mechanization: Since during the development process a much more abstract and also much more structured model is used, operations checking or manipulating these conceptual models can be defined. Those operations can be mechanized for full automation (like consistency checks) or to be guided or applied by the user (like module application, see below).

Besides the manipulation of the conceptual model - for example reusing specification modules [15] - the structured model also supports semantical analysis, for example model checking as described in section 7, as well as generation functionality, for example simulations or code as described in section 6.

6 Prototype Generation and Execution

Prototyping and simulation still are the prime tools for the validation and development of specifications. By visualizing runs of a prototype on the level of the description techniques used in the design of the system, properties of a system, for instance, behavior under specific environmental stimuli, can be conveniently observed and validated. 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.

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 multi-media visualization tools or external hardware systems to the simulation environment.

6.1 Code Generator

As described in [14] 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 the generated and compiled prototype code can be loaded dynamically into the simulation environment using Java's Class Loader. Second, Java offers a promising future as implementation language for embedded systems applications. A working prototype generator for Java available will then simplify the implementation of code generators for "real" embedded applications.

As a prime requisite for code generation the system specification must be consistent in the sense outlined in Section 4. The code generation is based on a synchronous semantics briefly outlined in Section 2.2.3 and in [14]. In the latter, where we describe the concepts of our prototyping approach, we suggested generating a Java class implementing the Java Interface *Runnable* for each component which would then be executed in its own, concurrent thread within the simulation environment. Enforcement of synchronity 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.

Obviously, in most cases it is not desirable to simulate a complete system under development, but only selected components thereof. To support this choice AutoFOCUS allows 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. However, users 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.

6.2 Interactive Animation and Visualization of Prototypes

Executing a generated prototype in a simulation is not very helpful if the simulation cannot be adequately visualized to developers. Usually, debugging and visualization of code execution are based on the program code. However, the generated 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 decomposition relationship. 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 (see Figure 8).

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 are attached to them.

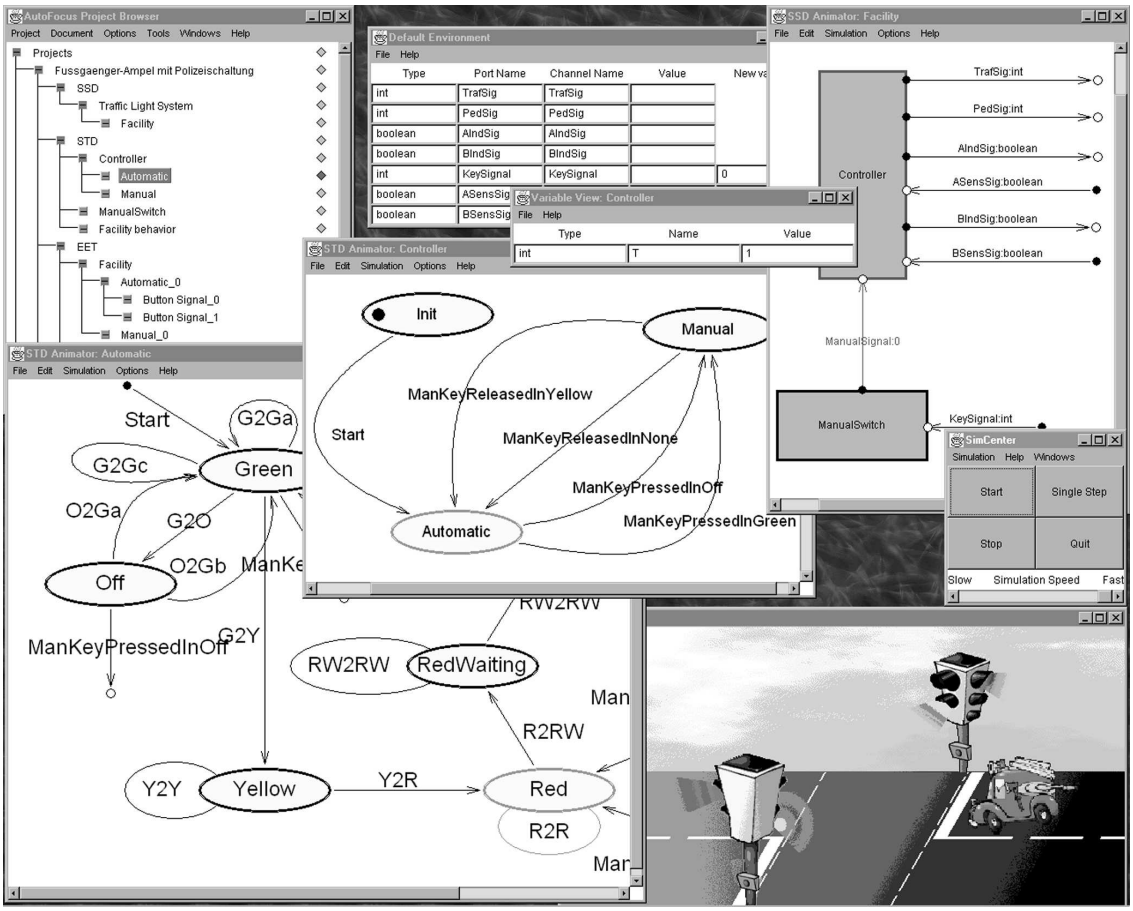

Figure 8: A Running Simulation Using Several Animators and Viewers as well as a FormulaGraphics-based Visualization

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.

6.3 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 than available by the simulation animators introduced in Section 6.2. Here, we understand "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, the 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 middle at the top of Figure 8 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.

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 front-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 *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, *FormulaGraphics* [10], and some "glue program code" to attach it to the SIMCENTER communication interface, the user-defined visualization shown also in Figure 8 (bottom right) 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 "pedestrian green phase" request. Since the design of this interface is generic, it can even be used to attach a real, hardware-based embedded system to the SIMCENTER environment.

6.4 Integration with Third-Party Tools via Code Export

In addition to the built-in Java code generator for simulation, a prototype of a code generator was implemented that generates code to be post-processed in external tools. Tabular and list representations of the elements in diagrams can be generated, as well as Perl [28] data structures intended to be further processed in Perl scripts. This way, a code generator that produces standalone C code from an executable specification has been implemented as well as generators for the input languages of a series of model checkers, such as SMV [9] or μ-cke [3]. The latter have then been successfully used to prove basic properties of our traffic lights case study (see Section 8.1).

7 Verification: Model Checking

With safety-critical embedded systems as its main application domain, AutoFOCUS aims at a far-stretched formal development. Thus, besides the proof of conceptual consistency of a specification and its validation using simulation, support for the semantic consistency is necessary. To support an engineering approach to system development, fully automated techniques like model checking are needed. As initially used for the clarification of the AutoFOCUS conceptual elements, the semantic foundation of the description techniques is now used to construct a sound mapping of the different system views onto the model used for model checking. In Subsection 6.4 we already introduced the interface of AutoFOCUS to external model checkers using the export interface. However, according to the integrated approach described in Subsection 1.2 an embedding of this proof technique transparent to the user is needed besides the external interface. As with simulation, all the information necessary for the verification can be expressed using the before-mentioned AutoFOCUS description techniques; furthermore, all necessary user interactions can be executed from within AutoFOCUS.

According to the structure of the AutoFOCUS description techniques and the system development approach (see Subsection 2.2 and 2.3, two forms of semantic inconsistencies can occur:

Inconsistencies of scenarios: Scenarios of a system as described by EETs cannot be executed by the realization of the system as described by SSDs and STDs.

Inconsistency of refinement: The execution sequences of a abstract component as described by an STD are not a superset of the execution sequences of the realization of the component as described by its sub-SSD and the associated STDs.

For both forms of semantic inconsistencies the subsequently described corresponding fully automated proof mechanisms are supplied.

7.1 Inconsistencies of Scenarios

Proving consistency of scenarios is of major importance during the early phases of a formal AutoFOCUS development process. Here, the behavior of a system, described by SSDs and STDs, is compared

with possible exemplary execution sequences described by EETs. If those exemplary sequences are used positively, an inconsistency occurs if those sequences cannot be executed by the described system, i.e., in case those exemplary execution sequences are not contained in the set of execution sequences of the system. Besides using them positively in form of requirement description, scenarios can also be used to formalize counterexamples to characterize unwanted behavior. Then, an inconsistency occurs, if an execution sequence described by the corresponding EETs is actually contained in the set of execution sequences of the system described by SSDs and STDs. Since EETs may describe both complete and partial behavior with further possible behavior not described within the EET, a proof mechanism is offered for complete and partial interpretation for both positive and negative interpretation.

7.2 Inconsistencies of Refinement

An essential design step in the development of a distributed system consists in the realization of an abstract component by concrete subsystem. In a formal development process as offered by AutoFOCUS, inconsistencies can be introduced in such a step if the behavior of the abstract component and of the concrete subsystem are both described. In AutoFOCUS, this is the case if an STD is assigned to a component as well as an SSD describing its substructure; furthermore, with each subcomponent, an STD has to be associated describing its behavior. An inconsistency occurs if the concrete realization can exhibit some form of behavior that is not possible with the abstract component. Formally this is the case if the set of execution sequences of the concrete realization is not a subset of the execution sequences of the abstract component.

7.3 Implementation

To check for inconsistencies of scenarios or refinement, the model checker for the relational μ calculus, μcke is used. The essential concepts of the formalization of SSDs, STDs and EETs are described in [16] and [15]. The user interface is structured using similar concepts as used with SIMCENTER in described in detail in [2]. Since model checking does not only check for consistency but furthermore supplies a counter example in case on an inconsistency, a mechanism for expressing those counter examples as EETs is currently under development.

8 Experiences with AutoFOCUS: Case Studies

Up to now, AutoFOCUS has been used to model a series of applications, the most important of which are briefly summed up in the following subsections.

8.1 Traffic Lights Controller

Several parts of this case study have been shown in the figures in the previous sections. For a complete description of this case study, we refer to [13]. Even for such a small system creating a correct specification is not a trivial task. Especially, unintentional specification of non-deterministic behavior of the system is a typical error, which is very difficult to locate by manual inspection of a specification. Here, the possibilty of SIMCENTER to indicate each non-deterministic execution step during the simulation was found to be of great value. Using the export interface mentioned in Section 6.4 input code for a number of model checking tools was generated from this specification, allowing to successfully prove selected properties of the specification. As expected, not all tested model checkers were equally suited for the execution model of AutoFOCUS, which is very similar to that of hardware circuitry.

8.2 Elevator Control System

The controller software for an elevator system was developed in a student term project. A central aspect was to test how well the SIMCENTER simulation could be connected to real hardware systems. Additionally to visualizing the simulation of the control software with a FormulaGraphics multi-media interface, a real hardware elevator model was controlled by the simulation. This project was demonstrated at

CeBIT'98. Unsurprisingly, the performance of such a Java-based simulation—although adequate for interactively monitoring the execution of the controller software—turned out to reach its limits when controlling real-time processes with "hard" time constraints in the sub-second range.

8.3 Production Cell

The specification of a production cell, see [23], is another case study done with AutoFOCUS. The production cell processes metal blanks using two presses and a two-armed robot. Besides the actuation units (belts, arms, presses, etc.), the system has several sensory units signaling the state of the robot arms, presses, tables, blanks on a belt, etc. Using the SIMCENTER interface to third-party systems, the simulated specification can be animated in a Tcl/Tk-based visualization developed at the "Forschungszentrum Informatik" in Karlsruhe.

8.4 Tamagotchi

In a comparative case study including other tools, such as StateMate, ObjectGeode, ObjecTime, and PEP, AutoFOCUS was used to create a specification of a Tamagotchi ([21]). The results of the study indicate that the integration approach of AutoFOCUS, in particular with respect to the description techniques and their expressiveness, was found to be very coherent and intuitive (despite the inherent weaknesses of a prototypical implementation), manifesting in a comparably short time required to get acquainted with the tool and its modeling concepts. Furthermore, multi-user support turned out to be a key feature for multi-developer projects. The possibilty to effortlessly generate prototypes, to execute them in a simulation environment, and to observe the simulation on the level of the modeling notations was found to be extremely useful both for rapidly locating specification errors and for trying out behavioral variants.

8.5 FM Banking System

AutoFOCUS in its QUEST variant including, for example, extended features for theorem proving and model checking, was applied to the example of the Formal Methods 1999 tool competition and acknowledged as the leading competitor among the participants. The features of AutoFOCUS/QUEST and more information on the case study can be found in [6].

9 Conclusion and Outlook

From the methodical point of view, the integration of a prototyping facility, SIMCENTER, into AutoFOCUS has proven to be an important step in helping developers to produce better system specifications. Executing a prototype of the system and visualizing its run using the description techniques used for modeling the system is a major benefit. Even further, embedding the prototype into an application-oriented visualization component can help developers enormously in rapidly locating errors in early specifications. Finally, such an application-oriented visualization of an embedded system offers a very suitable basis for communications between software engineers and application experts.

However, the possibility of validating a specification of an embedded system does not allow to *prove* that the system fulfills certain properties. Since this is mandatory especially in development of safety-critical systems, our future work on AutoFOCUS will further be focussed on an even tighter integration of formal verification tools such as model checkers and theorem provers.

The integration of methods for systematically testing specifications is another area of work that is becoming increasingly important in AutoFOCUS development. Questions addressing this field of research are treated in detail in the project QUEST, comissioned by the Bundesamt für Sicherheit in der Informationstechnik BSI. Furthermore, the above mentioned activities concerning the combination of specification tools and proof tools are performed in cooperation with QUEST.

A major benefit arising form the component-based paradigm used in AutoFOCUS is the support for reuse of components that are already developed and validated. This is, of course, possible within the conceptual framework behind AutoFOCUS, but not yet supported by the current version of the tool prototype

(see Section 2.4). The first, currently undertaken step is to migrate to a model-based repository implemented on top of an object data base.

References

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

[2] Bechtel, R. Einbettung des μ-Kalkül Model Checkers μcke in AUTOFOCUS. Master's thesis, Institut für Informatik, Technische Universität München, 1999.

[3] Biere, A. Effiziente Modellprüfung des μ-Kalküls mit binären Entscheidungdiagrammen. PhD thesis, Universität Karlsruhe, 1997.

[4] Bjorner, N, Browne, A, Chang, E, Colon, M, Kapur, A, Manna, Z, Sipma, H. B, and Uribe, T. E. STeP: The Stanford Temporal Prover (Educational Release) User's Manual. STAN-CS-TR 95-1562, Computer Science Department Stanford University, 1995.

[5] Booch, G, Jacobson, I, and Rumbaugh, J. UML Summary. Rational Software Corporation, January 1997.

[6] Braun, P, Lötzbeyer, H, Schätz, B, and Slotosch, O. Consistent Integration of Formal Methods. In Graf, S and Schwartzbach, M, editors, Tools and Algorithms for the Construction and Analysis of Systems, pages 48–62. Springer, 2000. LNCS 1785.

[7] Braun, P, Ltzbeyer, H, SchŁtz, B, and Slotosch, O. Consistent Integration of Formal Methods. In Graf, S and Schwartzbach, M, editors, Tool and Algorithms for the Construction and Analysis of Systems (TACAS 2000). Springer Verlag, 2000.

[8] Broy, M, Dederichs, F, Dendorfer, C, Fuchs, M, Gritzner, T, and Weber, R. The Design of Distributed Systems - An Introduction to FOCUS. TUM-I 9202-2, Technische Universität München, January 1993. SFB-Bericht Nr.342/2-2/92 A.

[9] Burch, J. R, Clarke, E. M, McMillan, K. L, Dill, D. L, and Hwang, L. J. Symbolic Model Checking: 10^{20} States and Beyond. Information and Computation, 98(2):142–170, June 1992.

[10] Formula Software. FormulaGraphics Multimedia System. Siehe auch http://www.formulagraphics.com, 1998.

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

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

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

[14] Huber, F and Schätz, B. Rapid Prototyping with AutoFocus. In Wolisz, A, Schieferdecker, I, and Rennoch, A, editors, Formale Beschreibungstechniken für verteilte Systeme, GI/ITG Fachgespräch 1997, pages 343–352. GMD Verlag, 1997.

[15] Huber, F and Schätz, B. Integrating Formal Descritpion Techniques. In Wings, J. M, Woodcock, J, and Davies, J, editors, FM'99 - Formal Methods, pages 1206–1225. Springer, 1999. LNCS 1709.

[16] Huber, F, Schätz, B, and Einert, G. Consistent graphical specification of distributed systems. In John Fitzgerald, Cliff B. Jones, P. L, editor, FME '97, LNCS 1313, pages 122–141. Springer, 1997.

[17] Huber, F, Schätz, B, Schmidt, A, and Spies, K. AutoFocus - A Tool for Distributed Systems Specification . In Bengt Jonsson, J. P, editor, Proceedings FTRTFT'96 - Formal Techniques in Real-Time and Fault-Tolerant Systems, pages 467–470. Lecture Notes of Computer Science 1135, Springer Verlag, 1996.

[18] i-Logix. Rhapsody Reference Version 1.0, 1997.

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

[20] Jones, M. P. An Introduction to Gofer, August 1993.

[21] Kamsties, E, von Knethen, A, Philipps, J, and Schätz, B. Eine vergleichende Fallstudie mit CASE-Werkzeugen für formale und semiformale Beschreibungstechniken. In Formale Beschreibungstechniken FBT'99. Utz Verlag, 1999.

[22] Lesny, C. Sprachbeschreibung und Parserimplementierung der funktionalen Sprache "Frisco F". Master's thesis, Institut für Informatik, Technische Universität München, 1997.

[23] Lötzbeyer, A. Task Description of a Fault-Tolerant Production Cell. Available via http://www.fzi.de/divisions/prost/projects/korsys/korsys.html, 1996.

[24] Rational. Rational Rose 98 Product Overview. Siehe auch http://www.rational.com/products/rose/, 1998.

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

[26] Selic, B, Gullekson, G, and Ward, P. Real-Time Object-Oriented Modeling. John Wiley and Sons, 1994.

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

[28] Wall, L and Schwartz, R. L. Programming Perl. O'Reilly & Associates, Inc., 1992.