Traffic Lights - An AutoFocus Case Study*

Franz Huber, Sascha Molterer, Bernhard Schätz, Oscar Slotosch, Alexander Vilbig
Institut für Informatik, Technische Universität München
80290 München, Germany
e-mail: (huberf|molterer|schaetz|slotosch|vilbig)@informatik.tu-muenchen.de

Abstract

In this paper we present a case study on AutoFocus*, a tool prototype for the development of distributed and concurrent systems based on the concepts of the formal method* Focus*. We develop (specify, consistency-check and simulate) the controller of a pedestrian traffic light using different graphical description techniques to illustrate an engineering process for concurrent systems.*

1 Introduction

The importance of software in embedded systems is rapidly increasing. More and more functionality of such systems, formerly realized using specialized hardware solutions, is now being implemented by software. This tendency will even increase in the years to come. Due to the very limited computing resources available in embedded systems controllers in the very early years, software development for embedded systems very much concentrated on the effort to tweak and optimize the code in order to cope with these circumstances. Today, high-performance micro-controllers allow increasingly complex software solutions for embedded systems. Thus it is obvious that the main focus in software development for these systems is shifting towards a task of managing the complexity and inter-relationsships of software components.

It is widely recognized that this increasing complexity of embedded software development can be managed only by adequate means of structuring, both from the point of view of the techniques used to describe different aspects of such systems and from the development process point of view. Both of these aspects, notations and process, should be supported by adequate tools. Due to the complexity of today's embedded software, facilities for verification and validation of properties of the software become a critical issue for efficient use of such tools in development.

AutoFocus is a tool prototype for development of concurrent embedded systems. Having its origins and its formal background in the formal development method Focus [1] it offers a structured approach to modeling embedded systems using graphical notations and a refinement-based development process.

Using the example of a pedestrian traffic lights controller, we demonstrate how a simple embedded system can be developed with AutoFocus.

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 focussing on simulation and code generation, like [2] or [3] and [4]. In general, however, these approaches 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 [5] includes several description techniques and offers tool support for the specification of systems using those techniques. However, UML lacks a precise semantical basis and therefore does not allow formal verification of vital system properties.

AutoFocus supports a lean subset of description techniques based on a common mathematical model, allowing both pragmatically oriented development and integration of formal techniques.

2 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),

*The authors of this paper were funded by DFG-Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen", the project "SysLab" supported by DFG-Leibnitz and Siemens Nixdorf, the project "KorSys" financed by the German Federal Ministry of Education and Research (BMBF), and the Forschungsverbund Software-Engineering (FORSOFT).

- 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 the 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 the user to switch between different levels of 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 (SSDs). Graphically, SSDs, as shown in Fig. 1, 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.

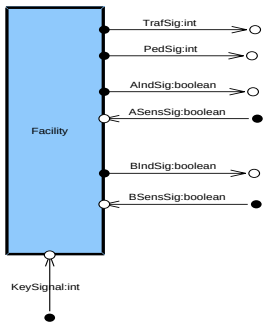

Figure 1: Black-Box Structure of Facility

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 a data type describing the set of messages that may be sent on it. In case no value is sent, the channel contains a default nil-value. Thus system structure diagrams provide both the topological view of a distributed system and the signature (the syntactic interface consisting of the ports of a component or system) 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 a SSD can be associated with

- substructures (SSDs),
- other 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.

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 [6][1]. The data types defined here may be referenced by other development 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 [7]. Graphically, STDs are represented by graphs with labelled oval nodes as states and labelled arrows as transitions. Fig. 2 shows a simple example.

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

- pre- and postconditions, formulated over the local data state of the component,
- 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
- an optional label, to replace the otherwise used conditions and patterns for better readability.

[1] In the actual implementation of AutoFocus we only use basic types which are also found in Java 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 Gofer into Java.

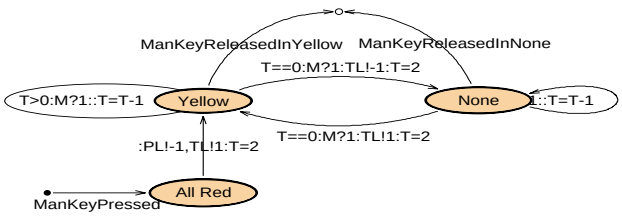

Figure 2: Behavior of the State MANUAL

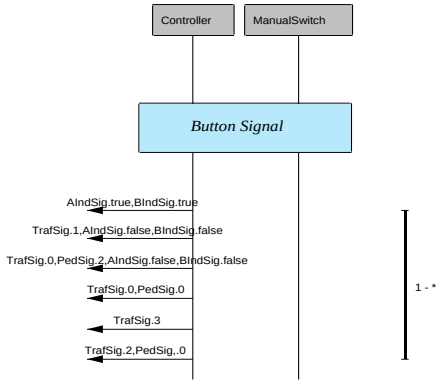

Figure 3: EET: Reacting on Button A or B

Like SSDs, STDs can be hierarchically organized by assigning an STD to a state. 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. This connector or port can be connected by edges from inside the substructures.

2.4 Extended Event Traces (EETs)

Besides STDs, extended event traces may also be used to describe dynamic 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 similar core concepts as found in [8]. As well as other graphical AUTOFOCUS notations, EETs support hierarchical concepts. Elements called Boxes can be inserted into an EET, referencing a set of sub-EETs grouped together. Since any behavior defined in this set may be substituted for the box, 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 [9].

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 meanings of DTDs and SSDs are quite obvious and need no further explanation, the exact meaning of the behavioral description technique of STDs has been 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 [10].

For the AUTOFOCUS description techniques, different but coherent semantic models exist: a stream-based semantics as described in [1] will be used to combine AUTOFOCUS with the *Isabelle* theorem prover, while a relational μ calculus semantics will be used for a model checking approach as described in [10],

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. 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 (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 channels and written to the output channels. An input pattern is a pair of an input channel 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 channel 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 an `!`. The expressions have to fit to the type of the channel, and may be built over free and local variables.

Consider, for example, the following transition as introduced in section 4.3:

```
T<0&&M==0;KS?W,ASS?true;AIS!true,BIS!true;
T=5,M=W
```

Here, `T` and `M` are variables defined for the component, while `W` is a free variable defined locally for this transition. Thus, `KS?W,ASS?true` will match if `true` is received on port `ASS`; the value received on port `KS` will be stored in `W`.

In preconditions only defined (local component) variables may be used to formulate predicates over the current data state. Since in the above example `T` and `M` are component variables `T<0&&M==0` is a legal precondition. 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. In the above example `T=5,M=W` is used as shorthand notation for `T'==5&&M'==W`, thus assign `5` to `T` and the read value of `W` to `M`. 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 interpreted 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 channel, the input pattern will only match if no message is received on this channel. Output messages are treated analogously. If no output pattern is defined for a channel, the value of the message written to this channel 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).

3 Informal Description: A Pedestrian Traffic Light

To demonstrate the AutoFocus development process, we use the case study of a pedestrian light controller as a running example. The case study is small, but it covers the central aspects of an embedded system, which makes it an adequate example for this article:

- It is an *open system*, since it communicates with the environment exchanging signals.
- It is a *timed system*, since the reaction of the system depends on the timing of the environment actions.[2].
- It is a *distributed* and *concurrent system*, since it consists of different components with independent control and communication by message passing.

It is also a safety-critical system, since unintended behavior of the system leads to possible major damage. In this section we give a informal description of the system, in Section 4.1 we develop a black-box interface view of the controlling system, and in Section 4.3 we refine it by a detailed specification of the system structure and behavior.

The engineering task consists of developing control software for a pedestrian light traffic controller guarding a pedestrian crossing. The pedestrian traffic light offers the following features:

- During the green phase for cars, a pedestrian can request a red phase by pressing a button located at the lights. An indicator light in both buttons will light up immediately confirming the action, and, after a short delay, the lights will be switched such that the crossing is free for pedestrians and red for cars.
- After a sufficient delay, the lights are set back to a green phase for cars, indicated by appropriate lights for cars and pedestrians.
- If no request is issued by a pedestrian for a certain time, all lights will switch off. A request in this state will be handled as in the first case.
- By activating the manual control, as soon as the system is back in the green phase for cars, the car lights can be set to a blinking mode while the pedestrian lights are switched off. Deactivating manual control will return the system to normal mode after a short all-red phase.

Figure 4 shows the crossing with the lights, buttons, indicator lights and the manual switch. Our task is to develop a safe realization of the system.

[2] We do not use the term *real-time*, since we have no requirements for continuous time, nor requirements for real response times.

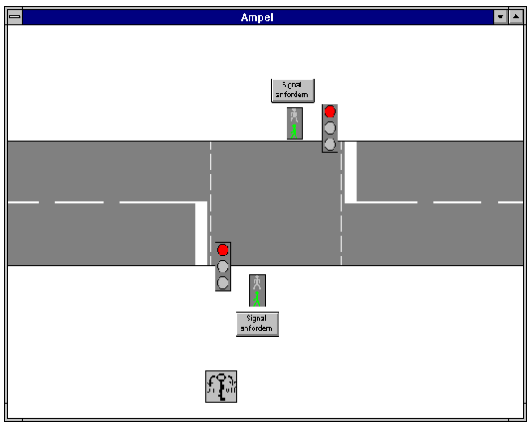

Figure 4: Simulated Pedestrian Crossing

4 Developing the System: The Controller

Having introduced the description techniques to work with and the application domain, we will now develop a suitable controller in a stepwise and incremental fashion. Each step will consist of four substeps:

1. *Interface and Structure Specification:* To specify the interface and structure of a system or component an SSD is used describing system structure and connections to the system environment. Furthermore, new data types may be defined using DTDs in case special message types are needed for those connections.
2. *Use Case Specification:* In general, positive or negative exemplary execution sequences of a component (or system) are specified using EETs, describing compulsory and illegal sequences, respectively. Again, such a description may be developed in a stepwise fashion itself, starting from abstract EETs using boxes and refining those boxes into a detailed execution sequence.
3. *Behavior Specification:* The complete behavior of a component is defined by assigning an STD to this component. Again, such a behavioral description may be developed in several steps itself, by first providing a more abstract STD and then refining some of its states. The behavior specification may include the definition of local variables using CDDs or even the introduction of new types for those variables by specifying new DTDs.
4. *Validation:* Finally, validation techniques may be used to relate this development step to the previous or to check the design decisions. Section 5 will treat this question in detail.

Not all of these substeps have to be present at every step in a development process, depending on future steps. For example, a step may only consist of the introduction of a new substructure of a component, delaying any behavioral specifications to a further refinement of the substructure.

4.1 System Interface

The specification of the system interface may also include the specification of of new data types by introducing new data constructs and corresponding operations on them. In this simple example only basic types (integer and boolean values) are necessary. Therefore no new data types are introduced.

The ports constitute the interface of a component or a complete system to the rest of the system or the environment, respectively. Those ports in an SSD not associated with a component are used as interface to the environment and are thus called "external ports". In the case of the traffic lights we have the following external ports of the complete system including the messages transmitted over them:

TrafSig: The signals to control the car lights are coded as values $-1, \ldots, 2$ (-1=Off, 0=Red, 1=Yellow, 2=Green).

PedSig: The signals to control the pedestrian lights coded as values $-1, 0, 1$ (-1=Off, 0=Red, 1=Green).

ASensSig, BSensSig: The signal sent from the buttons coded as boolean indicating whether a button is pressed.

AIndSig, BIndSig: The signal sent to light the indicator lights coded as boolean to switch lights on or off.

KeySignal: The signal sent form the manual control coded as values $0, 1$ for the right and left position of the key.

Since the car lights and the pedestrian lights in either direction always bear the same signal, we decided to use only one channel per each pair of lights. The buttons, their indicator lights and the key signal from the manual switch use separate channels. The buttons use boolean values, the key, and the lights are modelled by integers. The interface of the system is described in Fig. 1. In our example, the most abstract view of the system consists of only one component, called FACILITY. Therefore, each channel connecting the system to its environment is connected to one channel of the FACILITY component using a channel with the same name and type. In the refined system we will show two components connected to their

environment. Since ports are used to describe the syntactic interface of a component thus supporting modular system development, there is, of course, no need to have the same names for connected ports and channels. Equal names are only used for the reader's convenience in this example. The external and internal ports and the connecting channel of the traffic light system are listed in Fig. 11. Note that channel ManualSignal originates from a later refinement step. While the channel names will be used to describe communication actions with an EET, port names are used for send and receive actions in STDs.

4.2 Use Cases for System Behavior

Having fixed the interface between system and environment, we can now specify the behavior of the system on the chosen level of abstraction. We could already give a behavioral description by defining an STD, but we prefer to initially specify the behavior of the system with EETs. Such a definition can either be very detailed or consist of a small set of exemplary use cases outlining the system behavior. In our example we chose a medium approach, using hierarchy and indicators to describe a fair amount of use cases with "well-behaved" system actions: the manual key is only active during the cars' green phase, the buttons are only pressed if the indicator button is not lit (see Fig. 5, 6, and 7).

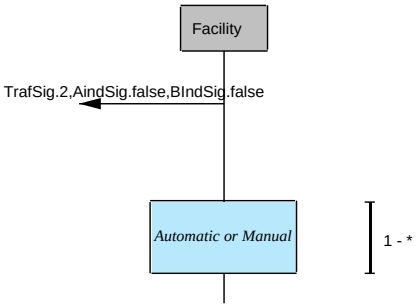

Figure 5: Root EET: Initialization

4.3 Complete System Behavior

After having defined the outline of the system behavior using EETs, we will now define the complete behavior using an STD. It will not only cover the complete behavior, but it will also specify the timing of the reaction of the system, something which we have not defined in the EETs.

Two local variables are used to specify the behavior:

Timer T: The timer variable is used to control delay loops for the delay between phase transitions. Therefore the lengths of the switching phases are controlled, like the length of the pedestrian green phase, or the time until switching off all lights.

Manual M: The manual variable is used to record the status of the manual switch, and thus to initiate the manual phase, if needed.

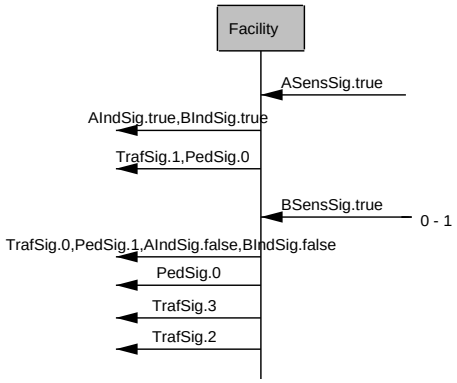

Figure 6: Sub-EET: Reacting on Button A

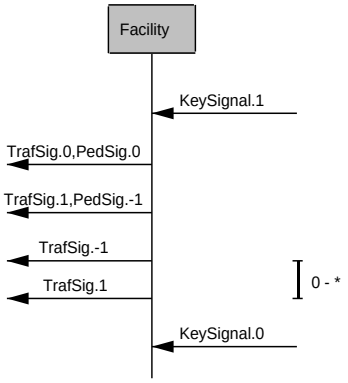

Figure 7: Sub-EET: Manual Mode

After adding those variables to the CDD of the FACILITY, the STD of the FACILITY is defined. Fig. 8 shows the non-hierarchical description of the system behavior. The local data definition of FACILITY is specified as `int T; int M`.

Even for a quite simple behavior such as the traffic light, the STD gets somewhat complex if no structuring mechanism is used. Using labels instead of the full transition semantics keeps the images readable, however an additional table (see Fig. 9) is required to explain the semantics.

In the refined system, we will show how hierarchy can be used to structure the description to obtain a

Label	State	Next	Pre	In	Out	Post
AR2ARKey	All Red	All Red	T==0	KS?W		T=T-1,M=W
AR2AR	All Red	All Red	T>0	KS?		T=T-1
AR2YBKey	All Red	YellowBlink	T==0	KS?W	PS!-1,TS!1	T=2,M=W
AR2YB	All Red	YellowBlink	T==0	KS?	PS!-1,TS!1	T=2
G2ARKey	Green	All Red	M==0	KS?	TS!0,PS!0	T=5
G2AR	Green	All Red	M==1		TS!0,PS!0	T=5
G2GaKey	Green	Green	M==0&&T<=0	KS?W,ASS?true	AIS!true,BIS!true	T=20,M=W
G2Ga	Green	Green	M==0&&T<=0	KS?,ASS?true	AIS!true,BIS!true	T=10
⋮	⋮	⋮	⋮	⋮	⋮	⋮
Start	Initialize	Green			TS!2,PS!0,AIS!false,IS!false	T=0
Y2RKey	Yellow	Red	T==0	KS?W	TS!0,PS!2	T=10,M=W
Y2R	Yellow	Red	T==0	KS?	TS!0,PS!2	T=10
YB2NKey	YellowBlink	None	T==0&&M==1	KS?W	TS!-1	T=2,M=W
YB2N	YellowBlink	None	T==0&&M==1	KS?	TS!-1	T=2
YB2RWKey	YellowBlink	RedWaiting	M==0	KS?W	TS!0,PS!0	T=10,M=W
YB2RW	YellowBlink	RedWaiting	M==0	KS?	TS!0,PS!0	T=10
YB2YBKey	YellowBlink	YellowBlink	T>0&&M==1	KS?W		T=T-1,M=W
YB2YB	YellowBlink	YellowBlink	T>0&&M==1	KS?		T=T-1

Figure 9: Transitions of the STD for Facility

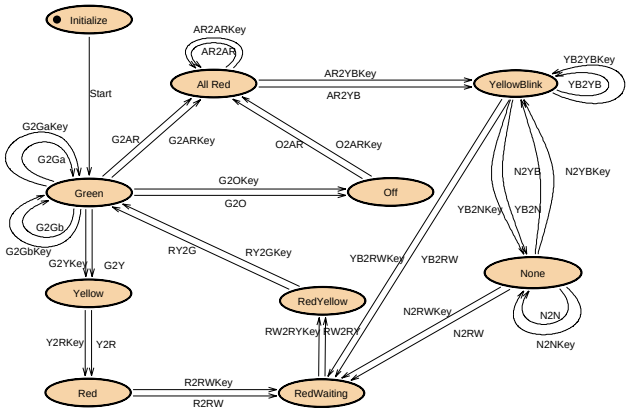

Figure 8: Behavior of Facility

more comprehensive specification.

To define an appropriate behavior as given by the informal description and the use cases introduced above, we introduce states for switching the traffic lights on a pedestrian's demand (`Green`, `Yellow`, `Red`, `All Red`, and `Red Yellow`), the sleep mode (`Off`), the manual mode (`All Red`, `Yellow`, `None`) and an initial state (`Initialize`, marked by a bullet). Furthermore, transitions between theses states with appropriate annotations have to be introduced. As example for the transition annotation we will examine the behavior given by the STD that is activated by a pedestrian pressing a button during the car green phase. The corresponding transitions are the feed-back loop transitions connected to state `Green`, for example the transitions labeled with G2GaKey, and G2Ga.

```
T<0&&M==0;KS?W,ASS!true;AIS!true,BIS!true;
T=5,M=W
```

This transition will be selected if no button has been pressed yet (`T<0` and the manual switch was not activated (`M==0`), some value from the manual switch is received (`KS?W`)[3] and button `A` is pressed (`ASS!true`). Executing the transition will light the indicator lights (`AIS!true,BIS!true`), set the timer (`T=5`)[4] and store the key signal (`M=W`). Since we expect the same behavior in case no key signal is present, we have to add

```
T<0&&M==0;KS?,ASS!true;AIS!true,BIS!true;T=5
```

to the previous transition.

This ends the specification of the traffic light controller in its "black box" form. Section 5 will treat the question of how this black box specification can be validated.

4.4 System Structure

The behavioral description of the system defined above using the STD could already be used as implementation of the controller. However, we are not satisfied with the complexity of the system, especially the treatment of the key signal, since it doubles the amount of transitions. Therefore, in the next step of this development process the controller of the complete facility will be broken up in a simple controller for the manual switch and a controller for the handling of the lights, buttons and indicators. The introduction

[3] Since `W` is a variable local to the transition, any defined signal on port `KS` will match.

[4] We use `T=5` as shorthand notation for `T'==5`.

of the manual switch controller eliminates the need to check for a key signal during the whole system execution - the manual switch status will only be checked in the Green phase for cars.

To refine the system structure, the SSD of the complete system is refined, using the hierarchy concept on the FACILITY component. An SSD describing its internal structure is defined, containing the manual switch and the controller as its components, as shown in Fig. 10. The external ports of this SSD are the ports of the refined component. As in the black box case, the external ports are connected with the ports of the components using channels of the same type. Figure 11 gives a detailed description of the connections of the glass-box structure of FACILITY, also including the ports.

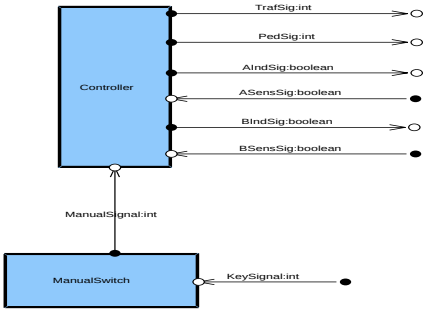

Figure 10: Glass-Box Structure of FACILITY

The internal ports, as shown in this table, will be used in the following specifications of the manual switch and the controller.

4.5 Uses Cases for Refined System

Like in the initial specification of the complete traffic light system, use cases can be defined for the refined FACILITY, too. Since we already provided a specification of the complete system behavior by the STD in 4.3, the definition of FACILITY is already fixed and no uses cases on this level of granularity are needed.

Channel	Type	Source	S-Port	T-Port	Target
AINDSIG	boolean	CONTROLLER	BAI	AIS	EXTERNAL
ASENSSIG	boolean	EXTERNAL	ASS	BAS	CONTROLLER
BINDSIG	boolean	CONTROLLER	BBI	BIS	EXTERNAL
BSENSSIG	boolean	EXTERNAL	BSS	BBS	CONTROLLER
KEYSIGNAL	int	EXTERNAL	KS	K	MANUALSWITCH
MANUALSIGNAL	int	MANUALSWITCH	M	M	CONTROLLER
PEDSIG	int	CONTROLLER	PL	PS	EXTERNAL
TRAFSIG	int	CONTROLLER	TL	TS	EXTERNAL

Figure 11: Channels and Ports in SSD of FACILITY

However, EETs can be introduced to give an outline of the behavior of each of the components. Similar as in the case of the complete system behavior, they can now be used as a starting point for the development of the refined system behavior described by STDs for the MANUAL SWITCH and the CONTROLLER. To specify use cases for the refined system, we "split" the "Facility" axis into a "Controller" and "Manual Switch" axis, and insert additional communication events between those two. Fig. 12, 3 and 13 show such refinements for the EETs introduced in Section 4.2.

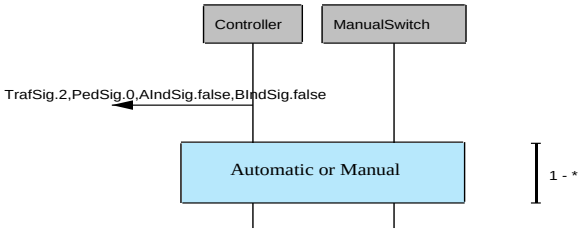

Figure 12: FACILITY EET: Initialization

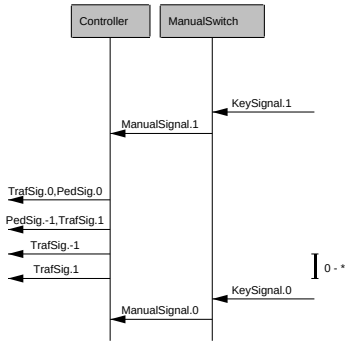

Figure 13: Sub-EET: Manual Mode

4.6 Complete Component Behavior

Since the refined traffic light system consists of two components, both considered to be sufficiently refined, we have to assign a complete behavior to both in order to supply a complete system behavior on this level of design. Therefore, STDs are defined for both of them.

While we need no local variables for the manual switch behavior, a timer variable as previously defined is needed for the controller. The following table lists the subcomponents of FACILITY and their CDDs.

Component	Local Data Definitions
CONTROLLER	`int T`
MANUAL SWITCH	

The STD describing the manual switch behavior, as depicted in Fig. 14 is quite simple. For each activation

state of the switch there is a corresponding STD state.

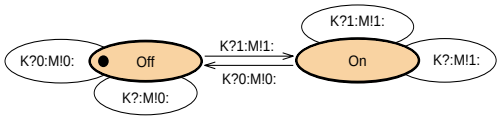

Figure 14: Behavior of the Manual Switch

The behavior of the controller is more complex. For better comprehension, it is described by a hierarchical STD, with the first level of abstraction shown in Fig. 15. For each of the main states of the controller,

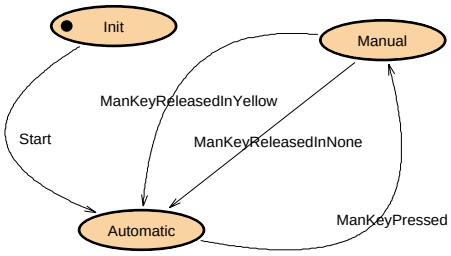

Figure 15: Behavior of the CONTROLLER

MANUAL and AUTOMATIC, corresponding sub-STDs are introduced. Note that each transition starting or ending at an abstract state will have a counterpart in the corresponding sub-STD. The semantics of the transitions are described in the table in Fig. 16.

Again, the detailed behavior of the state MANUAL as shown in Fig. 2 is quite simple. After entering through a short red phase, the states NONE and YELLOW are used for a blinking phase. Reactivating the automatic control is possible in both states: corresponding transitions leave the STD for this state and lead back to their counterparts in the abstract STD.

The STD describing the behavior in the case of automatic control is more complex. As mentioned above, it uses the ports described in Fig. 11 and the local variable `T` for the time dependent specification of the controller (see Fig. 17). For better readability of the documentation, AUTOFOCUS offers the possibility of displaying the transition annotations in a separate table, as shown in Fig. 18.

5 Validating the Model

After developing the specification of the traffic light system as described above, the next step in the engineering process is to validate whether this specification

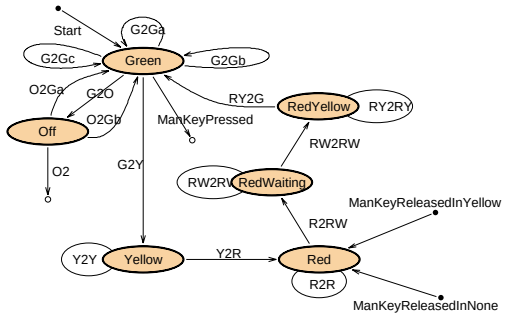

Figure 17: Behavior of the State AUTOMATIC

meets the initial requirements. So far, AUTOFOCUS offers two validation techniques for this purpose:

Consistency checks: Since AUTOFOCUS uses different as well as hierarchically organized views, it is necessary to ensure that the information - spread out over several views and probably edited by several developers - is well-defined or "consistent" in a methodological sense. For example, consistency conditions can be used to check whether each port is bound to a channel, or a specification is sufficiently detailed to be executable. AUTOFOCUS allows to check consistency conditions in one view or between different views. Finally, for flexibility reasons, the conditions themselves are not fixed. An AUTOFOCUS users can define new conditions using a declarative textual notation, similar to first order predicate logic with a simple type system.

Simulation: A consistent and executable specification can be simulated. The user can execute and step through the behavior that is defined in the STDs. SIMCENTER, the AUTOFOCUS simulation component translates the specification in Java code, which is then executed. As described below, the execution updates the views and thus supports graphical debugging of the specification. Especially in complex system this is an excellent way to detect specification errors, because the developer uses the same high-level description techniques both for specification and debugging. Furthermore, SIMCENTER allows to connect the simulated system to an application-oriented visualization component. In our example, we created a virtual pedestrian traffic light scene using a multimedia authoring tool as shown in Fig. 4 and animated it using the SIMCENTER simulation component. This offers a very suitable basis for communication between

Label	State	Next	Pre	In	Out	Post
ManKeyPressed	Automatic	Manual		M?1	PL!0,TL!0	T=5
ManKeyReleasedInNone	Manual	Automatic		M?0	TL!0,PL!0	T=10
ManKeyReleasedInYellow	Manual	Automatic		M?0	TL!0,PL!0	T=10
Start	Init	Automatic			TL!2,PL!0,BAI!false,BBI!false	T=-1

Figure 16: Transitions of the STD for Controller

Label	State	Next	Pre	In	Out	Post
G2Ga	Green	Green	T>-63	M?0		T=T-1
G2Gb	Green	Green	T<0	BBS?true	BAI!true,BBI!true	T=5
G2Gc	Green	Green	T<0	BAS?true	BAI!true,BBI!true	T=5
G2O	Green	Off	T==-63	M?0	TL!-1,PL!-1	
G2Y	Green	Yellow	T==0	M?0	TL!1,BAI!false,BBI!false	T=8
ManKeyPressed	Green	ExternalTarget		M?1	PL!0,TL!0	T=5
ManKeyReleasedInNone	ExternalSource	Red		M?0	TL!0,PL!0	T=10
ManKeyReleasedInYellow	ExternalSource	Red		M?0	TL!0,PL!0	T=10
O2Ga	Off	Green		BAS?true	BAI!true,BBI!true,PL!0,TL!2	T=5
O2Gb	Off	Green		BBS?true	BAI!true,BBI!true,PL!0,TL!2	T=5
O2	Off	ExternalTarget		M?1	PL!0,TL!0	T=5
R2RW	Red	RedWaiting	T==0		TL!0,PL!0	T=5
R2R	Red	Red	T>0			T=T-1
RW2RW	RedWaiting	RedWaiting	T>0			T=T-1
RW2RW	RedWaiting	RedYellow	T==0		TL!3	T=5
RY2G	RedYellow	Green	T==0		TL!2,PL!0	T=-1
RY2RY	RedYellow	RedYellow	T>0			T=T-1
Start	ExternalSource	Green			TL!2,PL!0,BAI!false,BBI!false	T=-1
Y2R	Yellow	Red	T==0		TL!0,PL!2,BAI!false,BBI!false	T=10
Y2Y	Yellow	Yellow	T>0			T=T-1

Figure 18: Transitions of the STD for Automatic

software engineers and application experts, since the latter are usually no experts in abstract description techniques.

5.1 Consistency

Before the specification for the traffic light system developed in Section 4 can be simulated, it has to be consistent. Even in such a simple example standard errors occur like

- A component/channel/port has no name
- A port is not bound to a channel
- A channel and an attached port do not have the same type
- The interface of a component does not match with its subcomponents

These consistency checks are not performed automatically during adding or changing SSD, STD or EETs, because developers should have enough freedom during the engineering process. If developers explicitly want to ensure the consistency of their specification, they can choose an appropriate set of conditions to check.

5.1.1 Defining Consistency Conditions

For our example, we add the condition "The interface of a component between its external and internal view must coincide." to the existing conditions. Fig. 19 shows the according AutoFocus dialog, in which the name, an informal description and declarative notation of the new condition are defined. Similar or

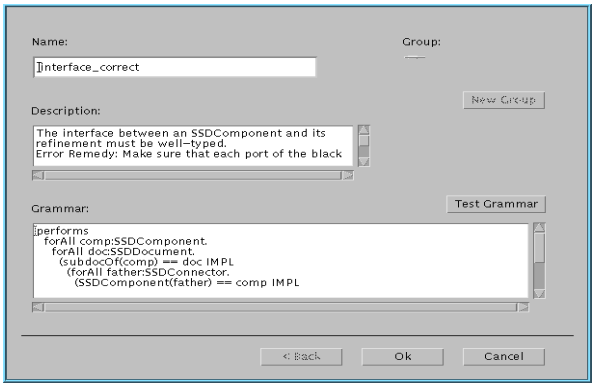

Figure 19: Adding the Consistency Condition "External and Internal Component Interface Must Coincide"

more complex conditions for other description techniques can easily be formalized using the declarative notation, because it offers appropriate individuals for the elementary objects like identifiers, components, channels, or directions and functions like `name_of`, `type_of`, or `direction_of`, as described in [10].

5.1.2 Checking Consistency

AutoFocus allows to apply consistency checks to a complete specification, to all views of a certain kind or to single views. All defined conditions are listed hierachically in a dialog. After the developer has chosen the desired conditions from this dialog, the result of the check is presented in a list to fix possible inconsistencies. Selecting an inconsistency from the list offers detailed information: the informal description as entered during the definition of the check is returned, and the universally quantified variables of the violated condition are returned. To fix the inconsistency, an immediate navigation mechanism leading to the corresponding elements in the specification is offered.

Consistency checks are getting more and more important as the size of the system or the number of involved developers increase. If a specification exceeds the toy world size of our pedestrian traffic light system, even simple consistency checks can be an enormous help.

5.2 Simulation

Once a (sub-)system has been specified consistently and detailed enough to be executable, it is possible to simulate it with AutoFocus using the built-in component SimCenter. The simulation as implemented by SimCenter addresses several vital issues of system development:

- Graphical debugging of a specification
- Prototyping by code generation based on the specification
- Problem domain oriented visualization of the system

5.2.1 Graphical Debugging

Debugging with SimCenter allows the developer to choose a (sub-)system, simulate it and observe the resulting system behavior through special views called *Animators*. In order to facilitate observation these views are animated versions of the views used to specify the system:

SSD Animators visualize the data flow aspects of a system during simulation. Channels on which messages are sent are highlighted and the values of the messages passed along them are displayed next to their names. **STD Animators** show the states and state transitions performed by components during the simulation. Current states and firing transitions are highlighted. **Variable Animators** display the values of state variables of components during the simulation. **EET Animators** show and record the communication history of selected components in a simulation. These communication histories provide a graphical runtime protocol of the simulation.

At any time during the system run the execution may be paused and continued, and every available animator window may be closed or openend. Thus the process of debugging is very similar to a conventional integrated development environment, greatly simplifying the search for logical faults in the system specification.

The process of simulating a specified system is illustrated by an exemplary simulation run: Referring to our case study example, we select the component Facility within the project as the simulated main component. Facility, however, is structured into subcomponents. Moreover both Facility and its two subcomponents Controller and ManualSwitch possess corresponding STDs within the project. Therefore SimCenter cannot decide for itself which specified behavior should be simulated and requires the developer to choose a (sub-)system to simulate. For this exemplary simulation run we select Controller and ManualSwitch[5].

After generating the executable code from the specification, SimCenter allows the user to start, stop and single-step through the simulation, as well as to open the available animator windows. Fig. 20 shows the EET animator view for the component Facility after several simulation cycles.

According to our specification, the input of the value `true` on the channel ASensSig indicates a pedestrian's request for a green phase. This request is recorded in the communication history of the EET animator for the component Facility as the first incoming message from the environment to the Controller. The next two outgoing messages send the value `true` on the channels BIndSig and AIndSig to the environment, thereby switching on both calling button indicator lights. After a few more simulation cycles the internal control state of Automatic consequently switches to `Yellow` and `Red` and the EET animator displays outgoing messages on the channels

[5] Note that both subcomponents have to be selected because only their combined behavior forms the behavior or their supercomponent Facility.

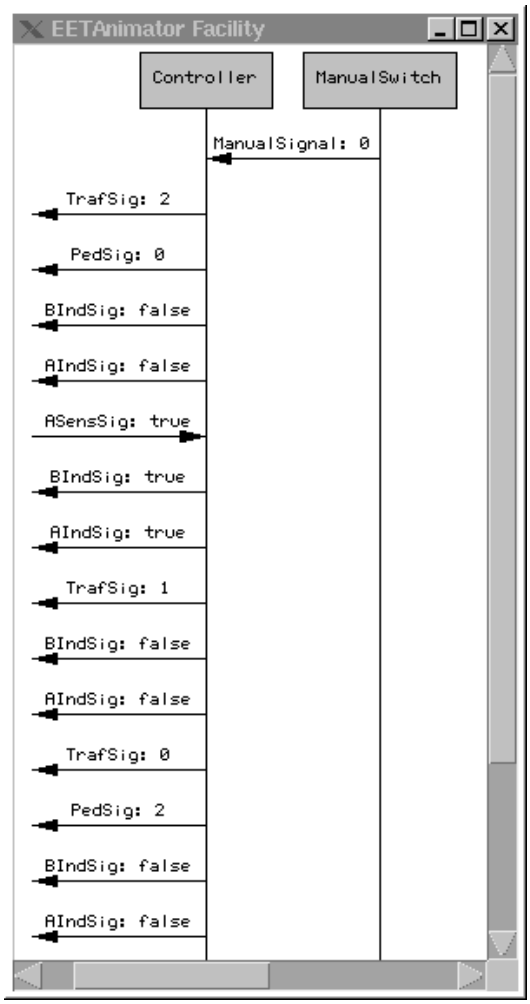

Figure 20: EET Animator View of FACILITY

TrafSig and PedSig commanding the traffic signal to switch from yellow to red (values 1 and 0 on channel TrafSig) whereas the pedestrian signal switches to green (value 2 on channel PedSig).

From the description of this exemplary simulation run, it is obvious that many simple logical faults of the specification, like "forgetting" to switch the traffic signal to red when the pedestrian signal switches to green, may be detected this way. Therefore the simulation in itself provides a valuable means to validate a system, although certain vital system properties, like "the pedestrian signal and the traffic signal never show green at the same time", require formal verification of the system.

5.2.2 Prototyping by Code Generation

The possibility to provide an executable prototype plays an important part in modern software engineering approaches. A prototype simplifies communication and understanding between customer and developer about the system to be developed. It is therefore much more likely that the finished product will fulfill the customer's expectations if a prototyping approach is used.

SIMCENTER generates Java code from the specification in order to simulate the system. Every component of the chosen (sub-)system whose dynamic behavior is specified with a STD gets translated into a single Java class with a very simple and human readable structure (cf. [11]). The original structure of the AUTOFOCUS specification is retained and the developer is therefore able to use the generated code as a basis for developing an executable prototype or even parts of the finished system. In many cases, however, it might be sufficient to use SIMCENTER's capabilities to attach external environments during a simulation run as will be described in the next section.

5.2.3 Application Oriented Visualization

The visualization techniques discussed so far, as represented by the different animators, are limited to an inside, developer oriented view of the system and its behavior. The observation of embedded systems, however, is very closely linked to the environment the system operates in. Customers are usually much more interested in a application oriented visualization of the system. Therefore SIMCENTER supports the attachment of external or even remote environments to complement the generated, rather simplistic default environment. These environments are specifically prepared for the system in question and are therefore able to present the running system in a problem domain oriented way. As an example of these possibilities we implemented an external environment using *Formula Graphics*, a freely available multimedia authoring tool, and visualized the pedestrian traffic light in a more familiar way. Actually, Fig. 4 is a screenshot taken from this interactive visualization. The user is able to click on one of the calling buttons labelled 'Signal anfordern' and after a specified delay the painted traffic lights will switch to red for the traffic and green for the pedestrians. A click on the button at the bottom of Fig. 4 symbolizing the police switch will lead to a constant flashing of the yellow light. This visualization of a traffic light is obviously much more vivid than the generated default environment.

The techniques used to prepare this exemplary environment are generally applicable and allow to visualize a broad range of different systems. However, it is also possible to connect other kinds of environments to SimCenter by using the provided interface. This general approach could be used to control and communicate with external hardware during the simulation. Thereby even the developer gains valuable knowledge about the running system within its future environment.

Graphical debugging, prototyping and visualization help to validate the system from two equally important views: Many simple logical faults may be eliminated by simulating and observing system behavior from a developer's point of view. Prototyping and problem domain oriented visualization facilitate to validate the system model from a customer's point of view. The need to verify certain safety critical properties, however, requires the use of a model checker or a theorem prover. Because the description techniques used in AutoFocus are based on a common formal semantics, it is our plan to integrate formal verification of a given specification in future extentions of AutoFocus.

6 Conclusion and Future Work

In this paper, we have shown an exemplary development process for a simple embedded system using the AutoFocus tool prototype. We demonstrated how systems specifications can be refined step by step up to a sufficiently detailed level of granularity. Our experiences with these techniques indicate that this approach seems adequate.

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. However, the possibility of validating a specification of an embedded systems does not allow to *prove* that the system fulfills certain properties. Since this is mandatory in certain areas like safety-critical systems, our current and future work on AutoFocus includes the integration of model checking techniques and tools as well as theorem provers.

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. Therefore, it also represents a major area of our current development activities.

References

[1] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T.F. Gritzner, and R. Weber, "The design of distributed systems - an introduction to FOCUS," Tech. Rep. TUM-I9203, Technische Universität München, Institut für Informatik, Januar 1992.

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

[3] D. Harel, "Statemate: A working environment for the development of complex reactive systems," *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 403–414, 1990.

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

[5] Grady Booch, Ivar Jacobson, and Jim Rumbaugh, *UML Summary*, Rational Software Cooperation, Jan. 1997, Version 1.0.

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

[7] Radu Grosu, Cornel Klein, Bernhard Rumpe, and Manfred Broy, "State transition diagrams.," Tech. Rep. TUM-I9630, Technische Univerität München, 1996.

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

[9] Bernhard Schätz, Heinrich Hußmann, and Manfred Broy, "Graphical development of consistent system specifications," in *FME'96: Industrial Benefit and Advances In Formal Methods*, Marie-Claude Gaudel and James Woodcock, Eds. 1996, Springer.

[10] Franz Huber, Bernhard Schätz, and Geralf Einert, "Consistent graphical specification of distributed systems," in *FME '97, LNCS 1313, pp. 122 - 141*, Peter Lucas John Fitzgerald, Cliff B. Jones, Ed. 1997, Springer.

[11] Franz Huber and Bernhard Schätz, "Rapid Prototyping with AutoFocus," in *Formale Beschreibungstechniken für verteilte Systeme, pp. 343-352*, A. Wolisz, I. Schieferdecker, and A. Rennoch, Eds. 1997, GMD Verlag (St. Augustin).

[12] Franz Huber, Bernhard Schätz, and Katharina Spies, "AutoFocus - Ein Werkzeugkonzept zur Beschreibung verteilter Systeme," in *Formale Beschreibungstechniken für verteilte Systeme*, U. Herzog H. Hermanns, Ed. 1996, pp. 165–174, Universität Erlangen-Nürnberg, In: Arbeitsberichte des IMMD, Bd.29, Nr. 9.