Heiko Lötzbeyer, Wolfgang Prenninger, Oscar Slotosch,
Ralf Steinbrüggen, Thomas Ströse, Ferdinand Winhard

AutoFocus-Tutorial

Institut für Informatik
Technische Universität München

Acknowledgements

The preceding introduction was compiled from a large number of publications and reports written at the teaching chair of Professor Broy in the course of the past ten years. We are particularly indebted to Mr. Franz Huber who did not only initiate AutoFOCUS and play a decisive role in its development for many years, but also made numerous texts and charts available for this tutorial. We are also grateful to Mr. Bernhard Schätz for his enthusiasm in the task of conceiving AutoFOCUS and for providing further resources on which this tutorial is based.

Contents

Chapter 1

Design Methodology

AutoFOCUS is a model-based tool for the development of reliable embedded systems. AutoFOCUS supports several stages of the development process (in particular the design stage), providing different graphical views of a component-oriented model. The simple component model can be used in many process models. The AutoFOCUS tool is particularly ideal for top-down development.

1.1 The Different Views of a System

The views of the model are an essential part of a model-based design. If we are dealing with graphical views, it is important (for reasons of clarity) to keep different aspects distinctly separate in different views. In addition, suitable hierarchy concepts are necessary to ensure that the diagrams have a manageable size. The component model of AutoFOCUS is hierarchical and several views have hierarchical parts themselves (e.g. substates in states). In AutoFOCUS, static and dynamic aspects of the system/the components are modeled in different views. We distinguish between four aspects or views of a system. Each modeling step concentrates on one of these views. For each view there is a special description technique:

- *structural view:* system structure diagram (SSD) to define the hierarchy of the components with their signatures, the ports, and the (typed, directed, synchronous) channels for exchanging messages.
- *interaction view:* sequence diagram, extended event traces (EET), message sequence charts (MSC) to represent the dynamic interaction of the components among each other and with their environment in the requirement analysis, during validation and verification.

- *behavioral view:* state transition diagram (STD) to define the behavior of the components by means of transitions with pre- and postconditions as well as patterns of the exchanged messages.
- *data view:* values and function definitions (data type definitions (DTDs)) to specify user-defined data types and auxiliary functions.

The descriptions of the four views of a system are integrated in AutoFOCUS. The consistency of the descriptions can be checked automatically. Simulation runs can be performed. The standardized descriptions facilitate the use of quality assurance tools. In the following chapter, we will deal with the description techniques in detail.

1.2 The Design Process with AutoFOCUS

AutoFOCUS supports several stages of the design process (see Procedure (see Chapter 9)), emphasizing a simple and clear modeling of the design. Particularly well suited to top-down design, the AutoFOCUS tool also supports the reuse of existing components (bottom-up). Figure 1 shows the typical sequence of steps. A system is first specified as a component which interacts with its environment and for which nothing but the interfaces to the environment are set. A few preliminary aspects of the behavior of the system can be defined by means of exemplary runs. Then the behavior can be stated definitely with a state transition diagram. If necessary, data to be administrated in the system can be declared.

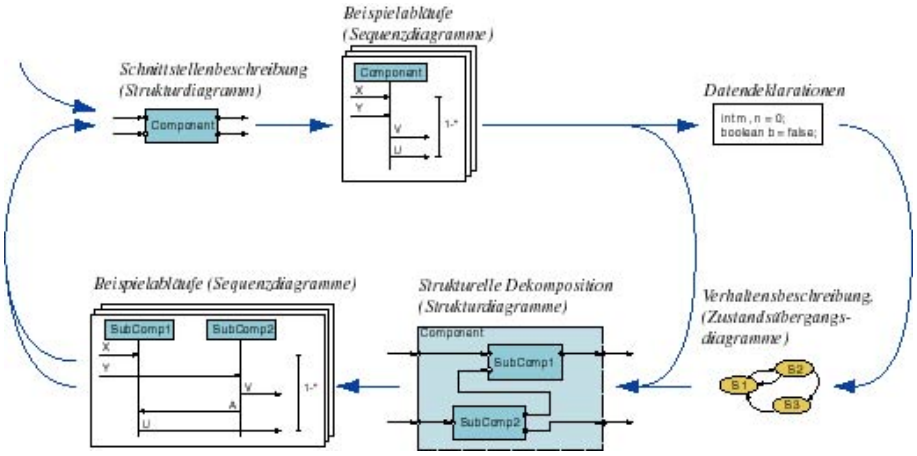

Figure 1.1: Figure 1: Overview of the system design with AutoFOCUS

When development begins, the system is first split up into a number of subcomponents, to each of which a more specific functionality is assigned. In order to enable these subcomponents to communicate with each other, they are linked by channels, creating a network. This process of a stepwise decomposition of the system may be repeated; the result is a hierarchical system structure whose granularity of detail will become finer with each iteration.

1.3 Tool and Applications

AutoFOCUS is an integrated tool for modeling, simulation, and validation. It facilitates the graphical visualization of different views of a system. There is an interface to further validation and verification tools. In addition, Validas AG has provided codegenerators for C, Java, Prolog, and Ada, and also offers professional support and consulting for AutoFOCUS. Numerous systems have been modeled using the AutoFOCUS design method and tool, including complex ones such as

- the control unit for the emergency closing system of the Eastern Scheldt storm surge barrier (see)
- the Mars lander (see)
- a control system for a production cell
- a smart card
- a car-seat controller
- a cash dispenser

Figure 1.2: Figure 2: The Eastern Scheldt storm surge barrier

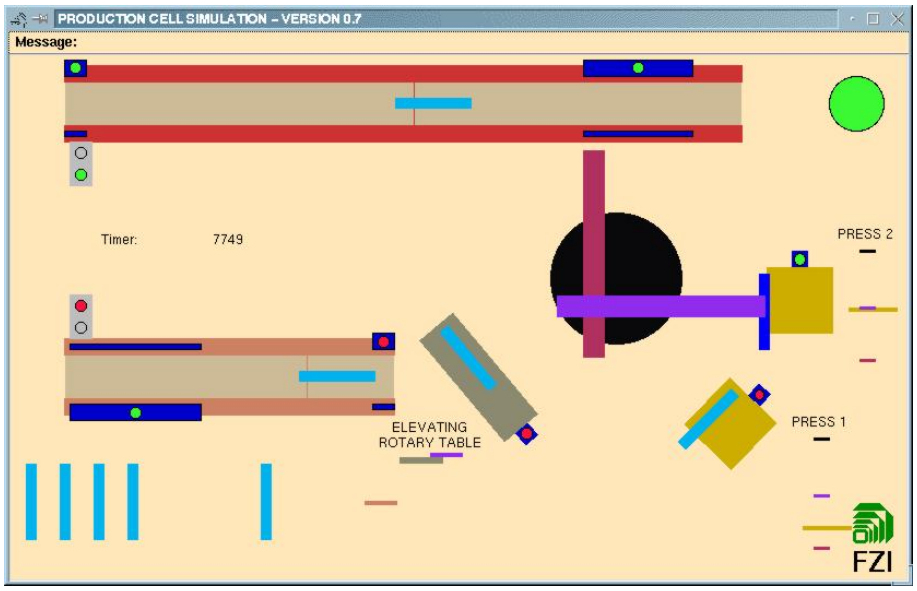

Figure 1.3: Figure 3: The production cell

Chapter 2

Description Techniques

In modern system development processes, different aspects of the system to be developed are described by means of different views of the system. The AutoFOCUS system design uses all the basic views outlined in the section ”Design Methodology” (structural view, interaction view, behavioral view, and data view). This results in four different partial designs whose interrelations are clearly defined. There are special means of description for each view.

2.1 The Traffic Lights Controller

In the following, we will develop a controller for a simple set of traffic lights with. It will serve as a running example to explain the various views of a system. The controller which will be developed as we are working through this tutorial can be simulated after complete definitions of the various layers have been retrieved. Traffic lights as shown in Figure 1 are frequently found at busy roads and near schools, railroad stations, and old people’s homes. On request, a green phase for pedestrians shall be activated, allowing them to cross the street safely.

2.1.1 Structure

The traffic lights consist of two poles, each with a stoplight for cars and one for pedestrians.

On each pole there is a push-button for requesting a green phase for pedestrians. Small indicator lights in each button indicate that one of the two buttons has been pushed.

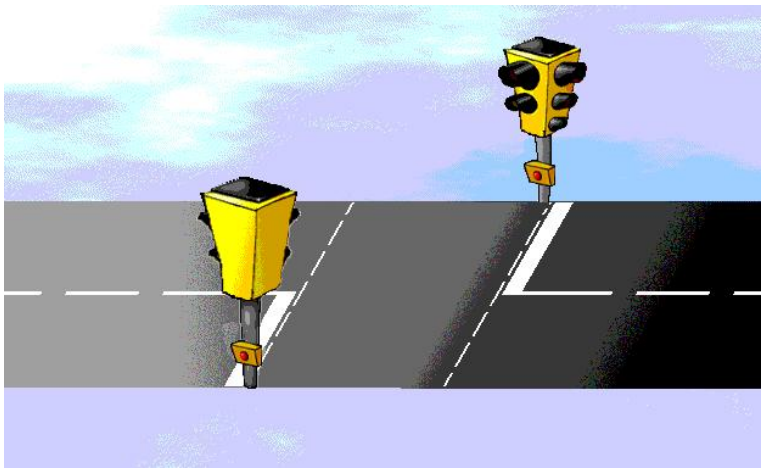

Figure 2.1: Figure 1: Set of traffic lights

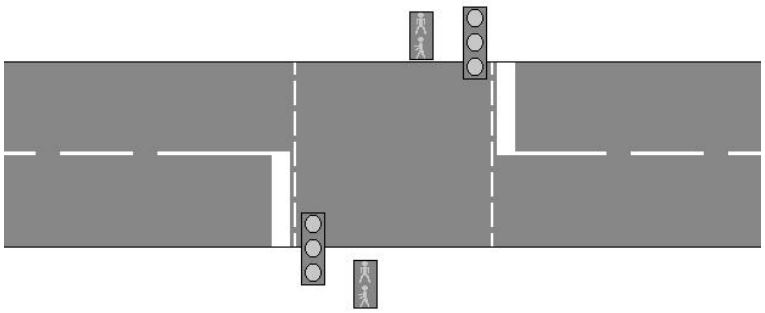

Figure 2.2: Figure 2: Schematic view of the set of traffic lights

2.1.2 Order of Events

As long as no green signal for pedestrians has been requested, the lights for pedestrians are ”red” and the lights for cars are ”green”. If one of the buttons is pushed, the lights for cars switch to ”yellow” after a predetermined period of time, and then to ”red”. Subsequently, the lights for pedestrians remain ”green” for a certain period of time. The indicator lights in the buttons of the pedestrian lights are switched off as soon as the lights for pedestrians are green. Next, the lights for pedestrians switch to ”red” again and the lights for cars first to ”red-yellow” and then to ”green.” During the whole process, the two lights for pedestrians and the two lights for cars respectively show the same signals at a time.

Chapter 3

Structural View

In this section, we will deal with a graphical description technique designed to capture the structure of information systems. The structural view focuses on the following questions:

- Which components does the system consist of?
- Which interfaces do the components have?
- Which connections (channels) exist between the interfaces?

3.1 Approach

Components are elements of a system which are physically or logically separated from each other. To make it easier to understand and describe a complex system, we split it up into several components. Each of the individual components may again consist of components. The individual components are not isolated from each other, but connected by *channels* via which they can exchange information The division of a complex system into various components and the network of channels connecting them is called *system structure*. In the system structure, directed channels are used through which data can flow only in one direction. There are various forms of diagrams to describe the system structure (the structural view). Most of them rely on the basic principle shown in Figure 3.1: components are symbolized by rectangular nodes and channels by arrow-shaped edges. The direction of the edges shows in which direction the information can flow. For example `Component 1` can read data from `Channel A` and `Channel C` and write data to `Channel B`. The data written to `Channel B` can then be read by

`Component 2`. Channels connected to a component only at one end have a connection to the environment at the other end, i.e. the system exchanges messages with its environment via these channels.

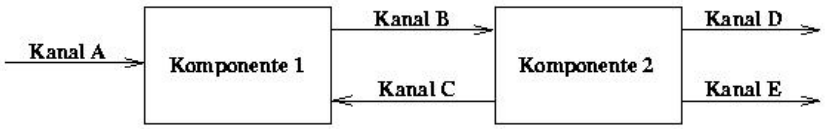

Figure 3.1: Figure 1: Structural sketch

3.2 System Structure Diagrams (SSDs)

To obtain a consistent graphical representation of the system structure, we use so-called *system structure diagram*(SSD). Figure 3.2 shows a SSD which provides a preliminary structural view of the traffic lights controller. The SSD contains only one component. It is called `Facility` and connected with its environment by several input and output channels.

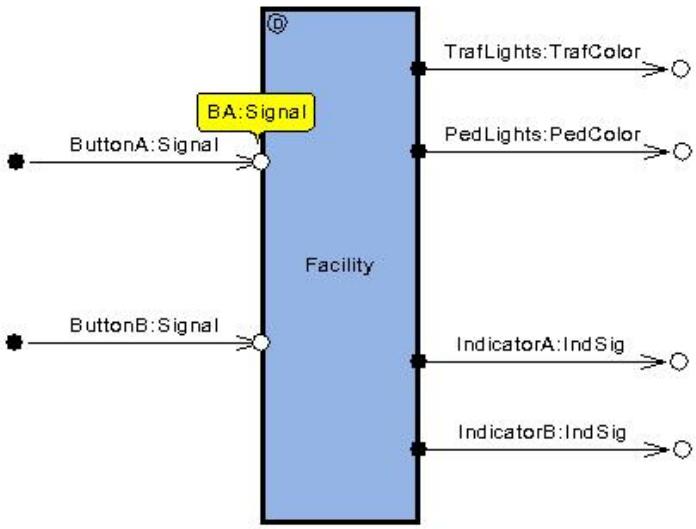

Figure 3.2: Figure 2: Interface view of the traffic lights controller

3.2.1 Channels

Channels are symbolized by labeled arrow-shaped edges. A channel's label consists of its name and type, separated by a colon. Which kind of information can be transmitted over a channel depends on its type. In the chapter "Data View" we will deal with data types in general, as well as with the special data types pertaining to our example. For the time being, it is just important to note that an unambiguous identifier must be chosen for each type.

For example, the edge labeled `ButtonA:Signal` symbolizes a channel with the name `ButtonA` and the data type `Signal` leading from the environment to the `Facility` component. The name `ButtonA` is meant to suggest that this channel connects button A, with which a green phase for pedestrians is requested, with `Facility`. This button A is considered a part of the environment. The same goes for the second button named B and channel `ButtonB`.

If a pedestrian pushes button A, a corresponding signal of type `Signal` generated and written to channel `ButtonA`. Now the `Facility` component can read and process this information. Next, corresponding information can be written to the `TrafLights`, `PedLights`, `IndicatorA` and `IndicatorB` output channels. This information is then transmitted to the environment, i.e. the lights for cars, the lights for pedestrians, as well as the two indicator lights.

3.2.2 Ports

So-called ports serve as interfaces between components and channels. In a SSD, they are symbolized by small circles at both ends of the channels. Empty circles correspond to input ports, solid circles to output ports. Just like the channels, the ports have names and types. For example, the input port on channel `ButtonA` has the name `BA` and the type `Signal`. Due to the introduction of ports, the interfaces of a component can be described independently of the channels attached to it. This has the advantage that components may be re-used in another context (i.e. in a system structure diagram with other channel names). In this case, one has to make sure that the types of connected channels and ports correspond. Otherwise, no data transfer would be possible between them. The names of channels and ports, by contrast, may differ. This enables components to be re-used as explained above.

3.2.3 Decomposition

As already indicated, a component itself can be regarded as an independent system as well. By gradually decomposing a component into subcomponents, the structure of a system can be increasingly refined. Each decomposition is represented in a new SSD and is also called a refined view of the system. Vice versa, the combination of various subcomponents into a component is called abstraction. If we only look at the interfaces of this component without its subcomponents and the channels connecting them, we also talk about an abstract view of the component (or the abstract component).

Figure 3.3 shows the decomposition of the `Facility` component in Figure 3.2. The small `D` in the upper left-hand corner of `Facility` indicates that a decomposition exists. By decomposing a component, we distribute its tasks among its subcomponents, thus reducing the complexity of the various views of the components used.

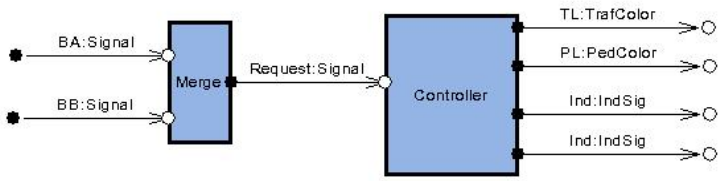

Figure 3.3: Figure 3: Decomposition of Facility (Ind is a port with two channels)

If the structural view is refined, ports serve as interfaces between the outside and the inside view of a component. All ports of the outside view (e.g. of `Facility`, Figure 3.2) appear as ports of the environment in the inside view (Figure 3.3). If we look at the component from the outside, only its interfaces or ports are visible. From outside, the interior of the component is hidden.

Basically, a channel always connects an output port with an input port. However, output ports may be connected to several channels, such as, for example, the `Ind` port attached to the `Controller` component. Channels connecting a component with the environment (such as `TL` in Figure 3) end in an input port of the environment as well. The environment, however, cannot be explicitly represented because it is not a constituent part of the refined view of a component or of the system. The input port to the environment in the refined view corresponds to an output port in the abstract view, since this is where signals come out of the component. In the same way, input ports of the abstract view of a component are linked to output ports of the refined view of a component. In our example, input port BA

in Figure 2 corresponds to output port BA (at the starting point of channel `BA`) in Figure 3.

In our example, each port of the environment is linked, via a channel, with one of the two components, `Merge` or `Controller`, in the inside view. The task of `Merge` is to merge the data from the two channels `BA` and `BB` in a single channel named `Request` and to transmit it to the `Controller` component. The `Controller` component is responsible for controlling the traffic lights.

System structure diagrams make it possible to describe the structure of a distributed system on different levels of abstraction. The diagrams cannot show the actual behavior of a system. For this purpose, we use extended event traces (EETs) and state transition diagrams (STDs) respectively. For these, however, a description of the structure is required, as will be seen in the chapters ”Interaction View” and ”Behavioral View”.

3.2.4 Summary

- SSDs describe the structure of a system with the aid of components, channels, and ports.
- Ports are the interfaces of the components.
- Components communicate with each other via directed channels.
- Ports and channels are typed in order to determine which kind of data is sent through them.
- A hierarchical description of the system is obtained by means of decomposing the components.

3.2.5 Procedure

When modeling begins, the system as a whole is usually represented by just one single component. First, we have to determine which pieces of information enter and leave the system and in which form. From this result the input and output channels to the environment (Figure 2). The SSD obtained in this way is also called the interface view of the system.

In the next step, we refine the structure. This means, in particular, that subcomponents are identified, the tasks are allocated, and the channels between the subcomponents are specified. The result is called decomposition (Figure 3). If necessary, individual components are decomposed further. The view obtained in this way is

also referred to as architecture view. When you create the system structure diagram of the traffic lights example yourself in the chapter ”Instructions for SSDs”, you will have the opportunity to re-enact this procedure once again.

3.2.6 Structural View of the Traffic Lights Example

Embedding into the Environment

In the first step, we are exclusively concerned with the embedding of the traffic lights controller into its environment without distinguishing between individual components. The ”traffic lights controller” system as a whole is represented by one single component called `Facility`. We exert influence on the controller by means of the two push-buttons for requesting the green phase for pedestrians. Their states are registered via the two input channels `ButtonA` and `ButtonB` respectively. The control unit in turn influences the lights for cars via the `TrafLights` output channel, the lights for pedestrians via the `PedLights` output channel, and the two indicator lights via the output channels `IndicatorA` and `IndicatorB` respectively. In addition, a suitable type is assigned to each channel. (Data types will be dealt with in the chapter ”Data View”). Figure 3.4 shows the SSD we are aiming at.

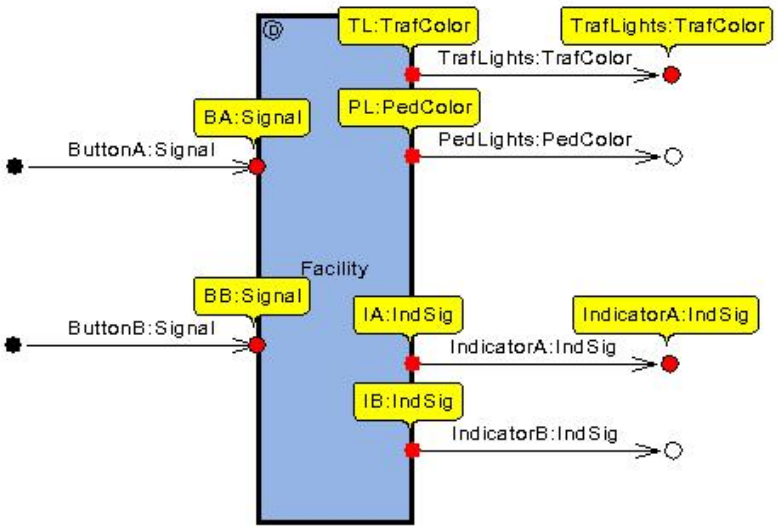

Figure 3.4: Figure 4: Interface view of the traffic lights controller

The ports of the `Facility` component are named with abbreviations according to the channels the ports are associated with. For example, the name of output port `TL` is derived from the `TrafLights` channel. The labeled balloons indicate a port's name and type.

Decomposition

In the next step, we refine `Facility` into two components named `Merge` and `Controller`. Figure 3.5 shows the decomposition of `Facility` into two components. The small D in the upper left-hand corner of `Facility` (Figure 3.4) indicates that a decomposition exists.

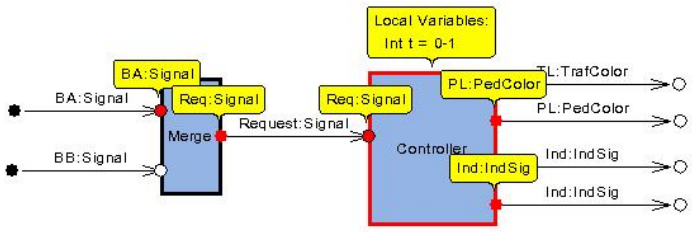

Figure 3.5: Figure 5: Architecture view of the traffic lights controller

The `Merge` component waits until, upon the touch of a button, a date is sent on one of the two channels `BA` or `BB`. `Merge` forwards this date unchanged via its `Request` output channel. In other words, `Merge` merges the data of channels `BA` and `BB` on one channel named `Request`. `Controller` receives requests for a green phase over this channel and controls the lights for cars, the lights for pedestrians, and the two indicator lights via the output channels `TL`, `PL` and `Ind` respectively.

3.3 Instructions for SSDs

In the following, we will explain step by step how to create the system structure diagram for our traffic lights controller in AutoFOCUS. Our aim is to create a complete structural view of the traffic lights example. See also:

- Informal Description of the Pedestrian Traffic Lights
- Introduction to the Structural Description
- System Structure Diagrams

View	AutoFOCUS Description Technique	Abbreviation
Structural View	system structure diagram	SSD
Data View	data type definition	DTD
Interaction View	extended event traces	EET
Behavioral View	state transition diagram	STD

Table 3.1: Sichten in AutoFocus

3.3.1 New Project

Start AutoFOCUS. The starting process is described in the installation instructions. A window with the project browser is displayed, which does not yet contain any project entries when used for the first time. As the windows look different depending on the operating system used, they may differ from the examples shown here. First, we create a new project for modeling the traffic lights controller. For

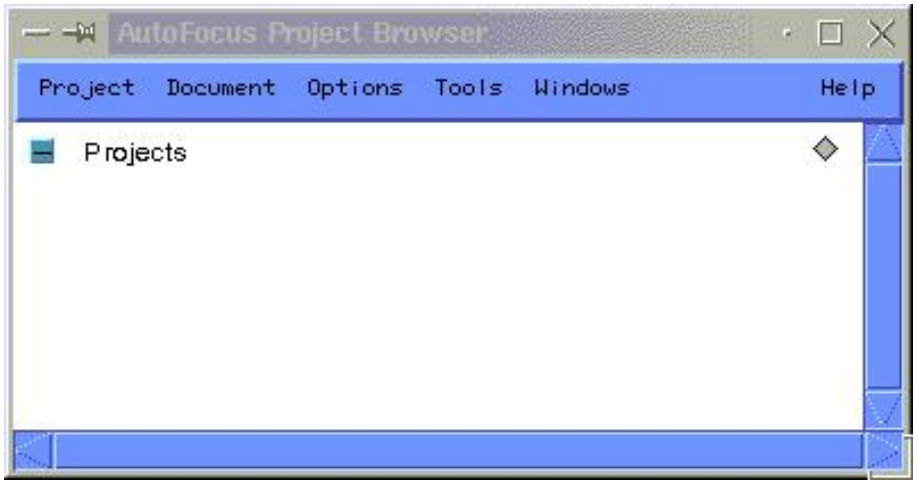

Figure 3.6: Empty project browser

this purpose, select the ”New Project” item from the ”Project” menu. Type the name *Pedestrian Traffic Lights* in the text field, keep ”Quest” as the data format of our new project, and finish by clicking ”Ok”. Our ”Pedestrian Traffic Lights” project is now displayed under ”Projects” in the project browser. If you click on the plus button, the four views of the project are displayed which AutoFOCUS offers:

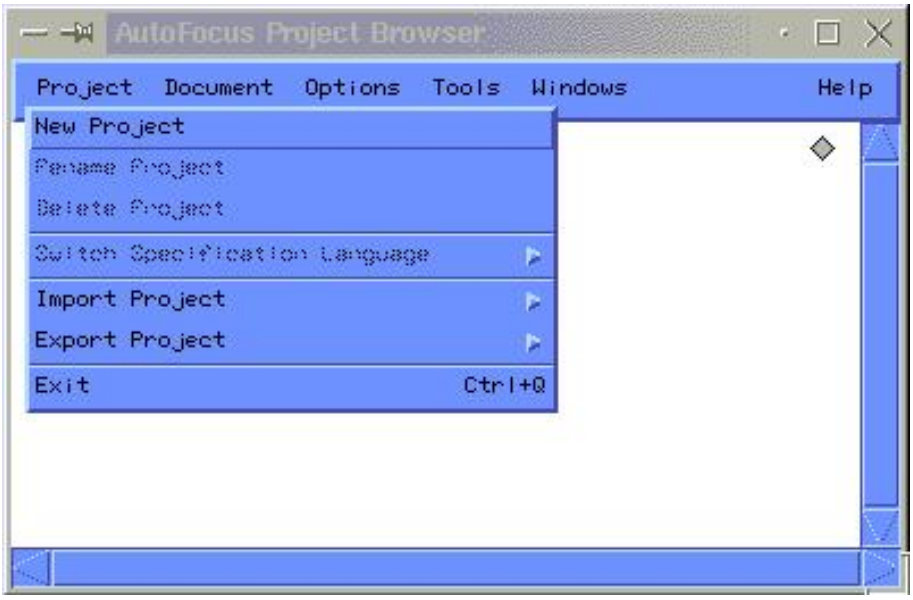

Figure 3.7: ”Project” menu in the project browser

Figure 3.8: ”New Project” dialog window

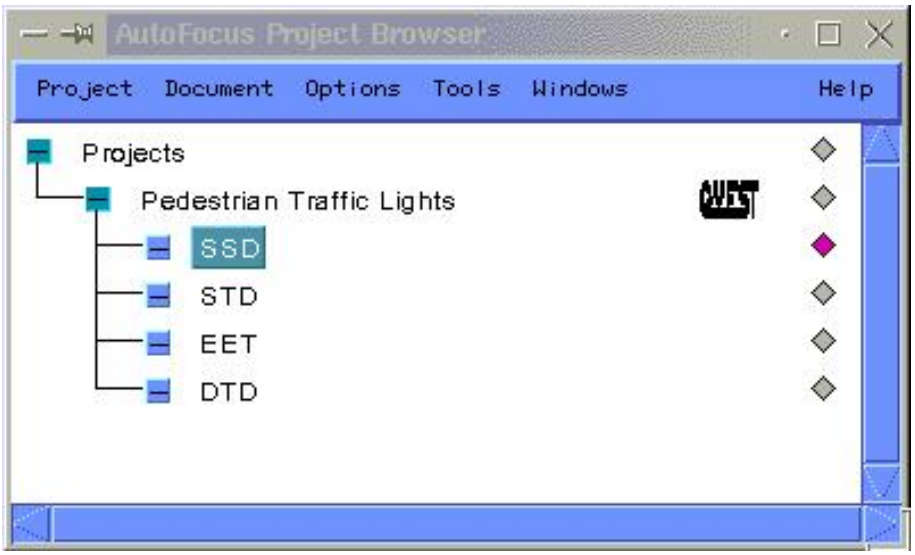

Figure 3.9: Project browser with the ”Pedestrian Traffic Lights” project

3.3.2 New SSD

To create a new system structure diagram, mark the ”SSD” entry in the project browser (cf. the figure above) and select the command ”New” from the ”Document” menu.The dialog window ”New SSD” is displayed. Name the new SSD *Mainsystem* and leave the dialog window by clicking ”Ok”. In a new window, AutoFOCUS now starts the SSD editor for editing the structure of `Mainsystem` It contains a menu bar and an icon bar with four buttons.The buttons are used to select the cursor function. The following table shows the functions of the buttons (from left to right):

Symbol	Name	Function
	Select	Selects elements of the drawing (highlighting them red).
	Component	Creates a new component.
	Channel	Creates a new channel.
	Delete	Deletes diagram elements by clicking them with the mouse.

Figure 3.10: ”New SSD” dialog window

General advice on operating the editors in AutoFOCUS can be found in the operating instructions for editors.

3.3.3 New Component

To create a new component in our SSD, select the ”Component” function by clicking on the second button from the left. Click with the left mouse key on the SSD editor’s white working space at the spot where you want the new component to be situated. A ”Set name” dialog window opens, asking you to enter the name of the component. Type *Facility* and confirm by clicking ”Ok”. To change the component’s size, activate the ”Select” button in the icon bar and drag the edges or corners of the component into the proper position while holding the left mouse key down. The component in queation must be clicked in advance to be active. The component as a whole can be relocated by holding down the mouse key inside the component and moving the cursor (also see the operating instructions for editors).

New Channel

Now we model connector channels from the environment to the component (input channels of the component) and from the component to the environment (output

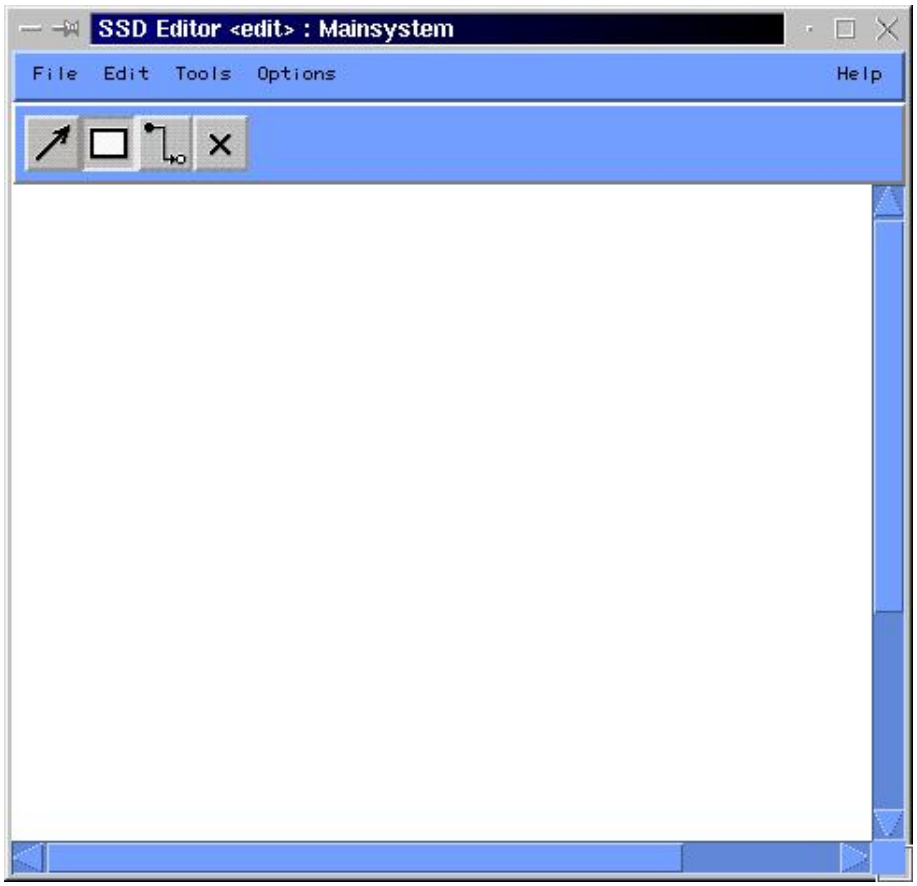

Figure 3.11: Empty SSD editor

Figure 3.12: ”Set name” dialog window

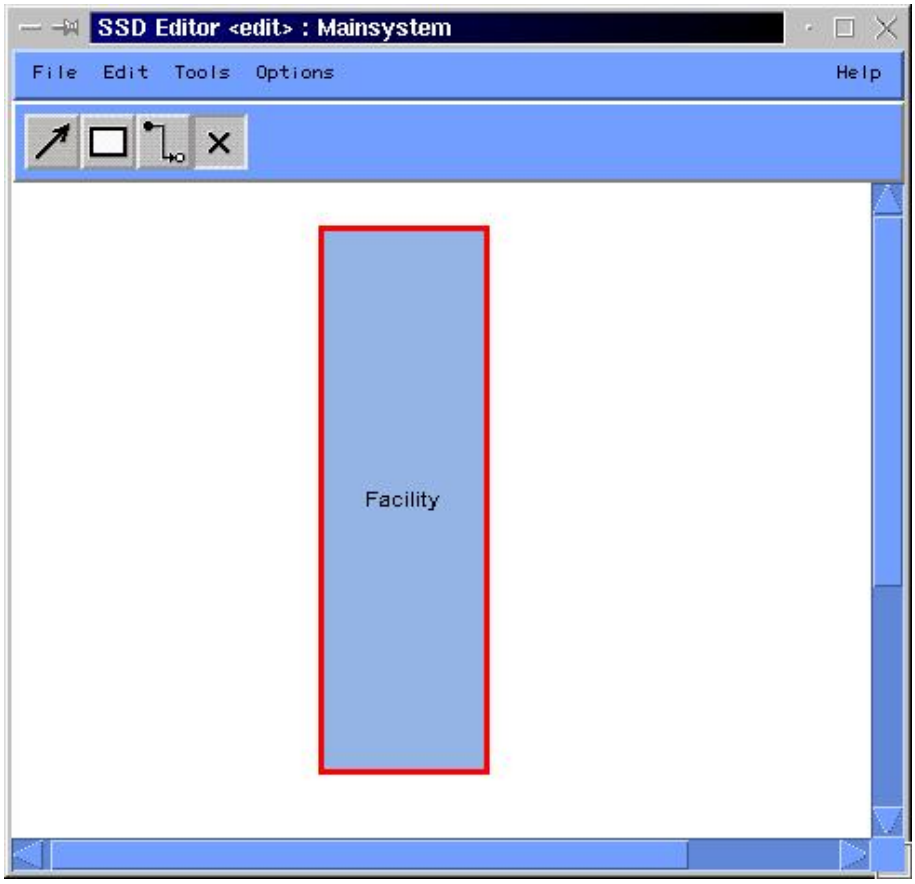

Figure 3.13: SSD editor with the Facility component

channels of the component) respectively. For our example, we create two input channels coming from the left and four output channels leading to the right:

To create the input channels, activate the ”Create Channel” button and drag the cursor horizontally from the left outside the component (white space) into the component (blue space). A dialog window opens, in which you can enter the channel name and the appropriate channel type: Call the first channel *ButtonA*,

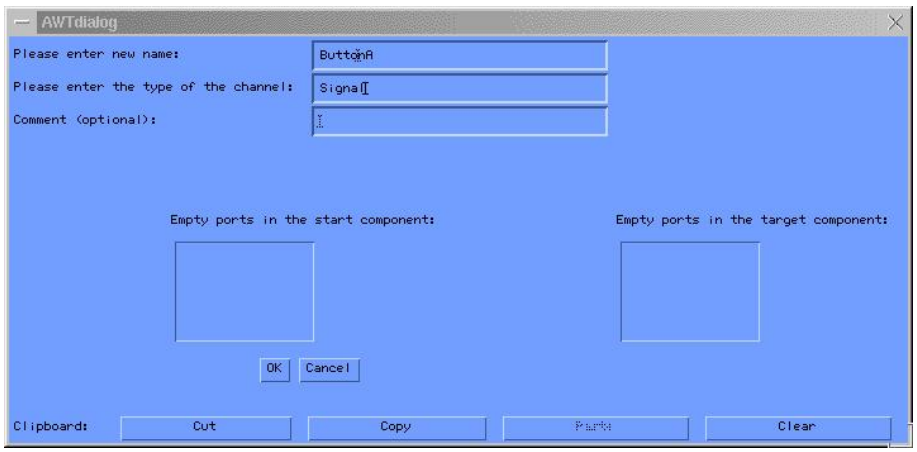

Figure 3.14: ”New Channel” dialog window

set its type to *Signal* and finish your entry by clicking ”Ok”. Analogously, you create the second input channel and call it *ButtonB*. Now the SSD should look approximately like the left part of the figure below: In a similar manner you now create the four output channels and name them as shown in the figure above. For this purpose, click inside the ”Facility” component, hold the mouse key down, drag to the right and release the mouse key outside the component.If several similar channels are required, you may want to make use of the *clipboard* to enter the channe data: for example, after you have entered the name and type of channel ”IndicatorA” into the mask, you copy the entries into the clipboard by clicking on the ”Copy” button before finishing your entry with ”Ok”. Next, you draw channel ”IndicatorB”. However, instead of entering the name and type into the text fields all over again, all you have to do now is to insert the clipboard content in the mask by clicking on ”Paste” and then change ”A” to ”B”.

3.3.4 Ports

If a channel is created, its associated input and output ports are generated automatically. The ports’ names and types are adopted from the associated channel.

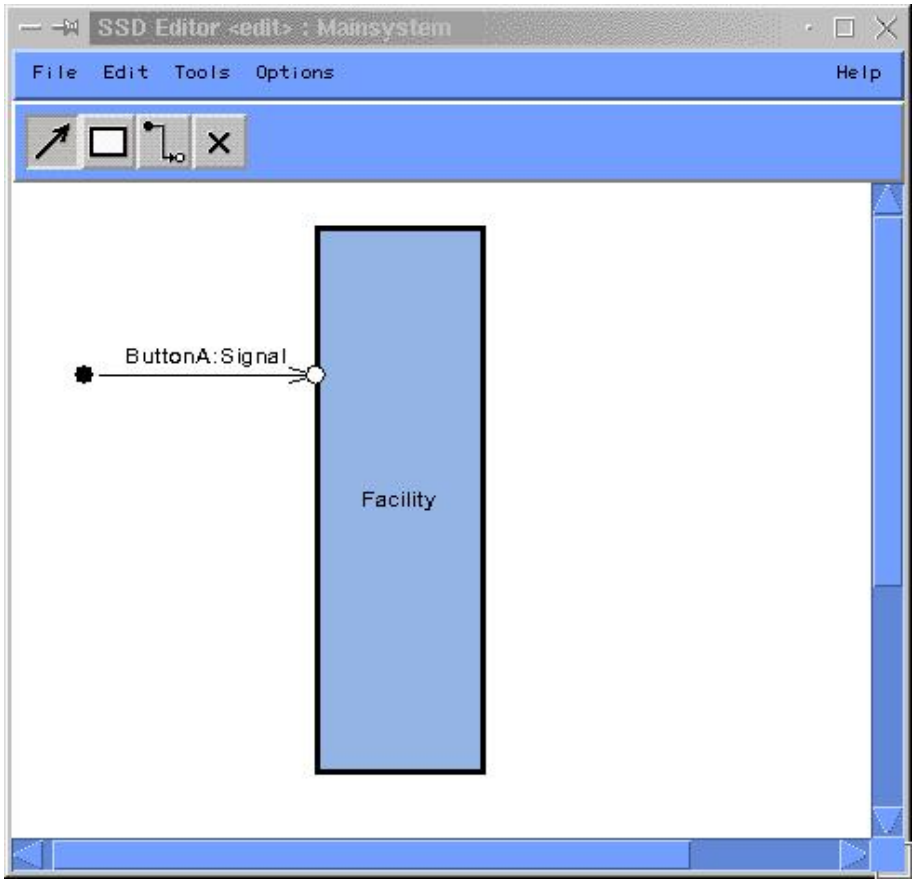

Figure 3.15: SSD editor with channel ”ButtonA”

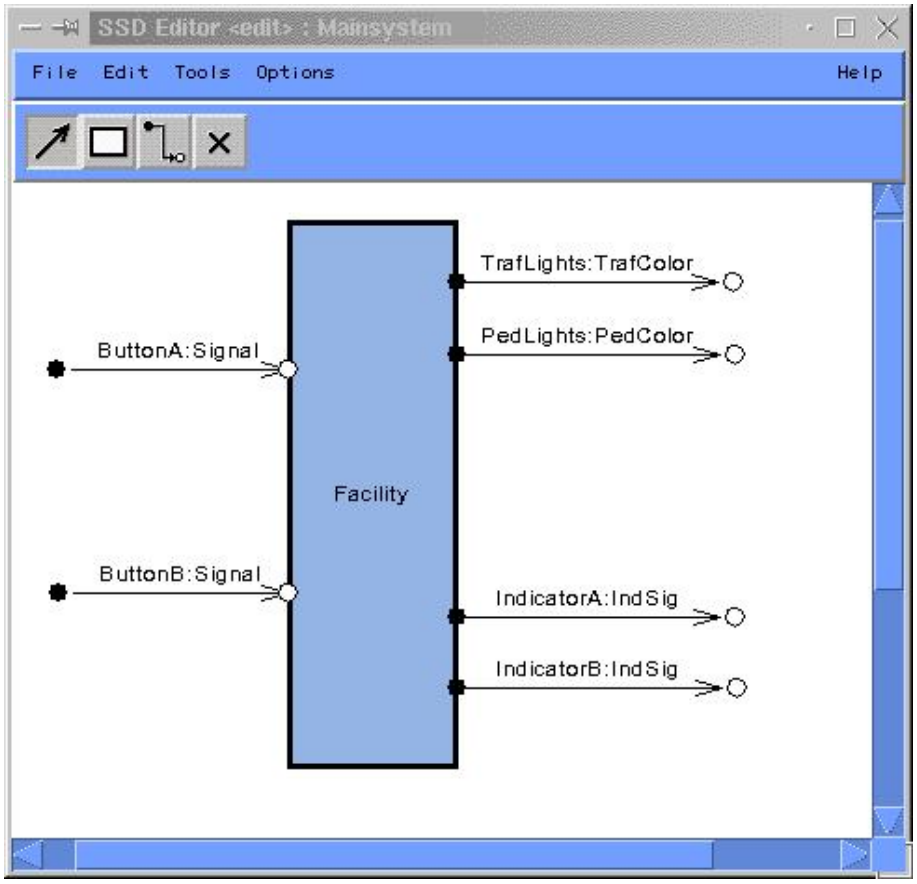

Figure 3.16: SSD editor with 6 channels

Now we rename the two input ports and the four output ports of ”Facility”:

- Click on the ”Select” button
- Select the port (the port is now highlighted red)
- Edit menu - select attributes (instead of following the latter two steps, you also have the option of ”double-clicking” the respective port)
- The ”Port Attributes” dialog appears. Change the port names in accordance

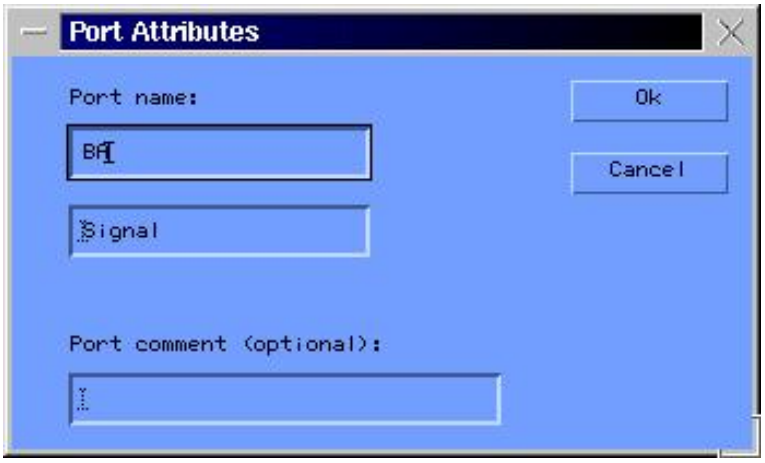

Figure 3.17: ”Port Attributes”dialog

with the following table:

Previous Port Name	New Port Name
ButtonA	BA
ButtonB	BB
TrafLights	TL
PedLights	PL
IndicatorA	IA
IndicatorB	IB

The figure below shows our component with the renamed ports. To select several ports at the same time, click on them with the shift key pressed.

3.3.5 Decomposition Step

Next, we refine our description (specification) of the traffic lights controller in compliance with the structural view of the traffic lights example::

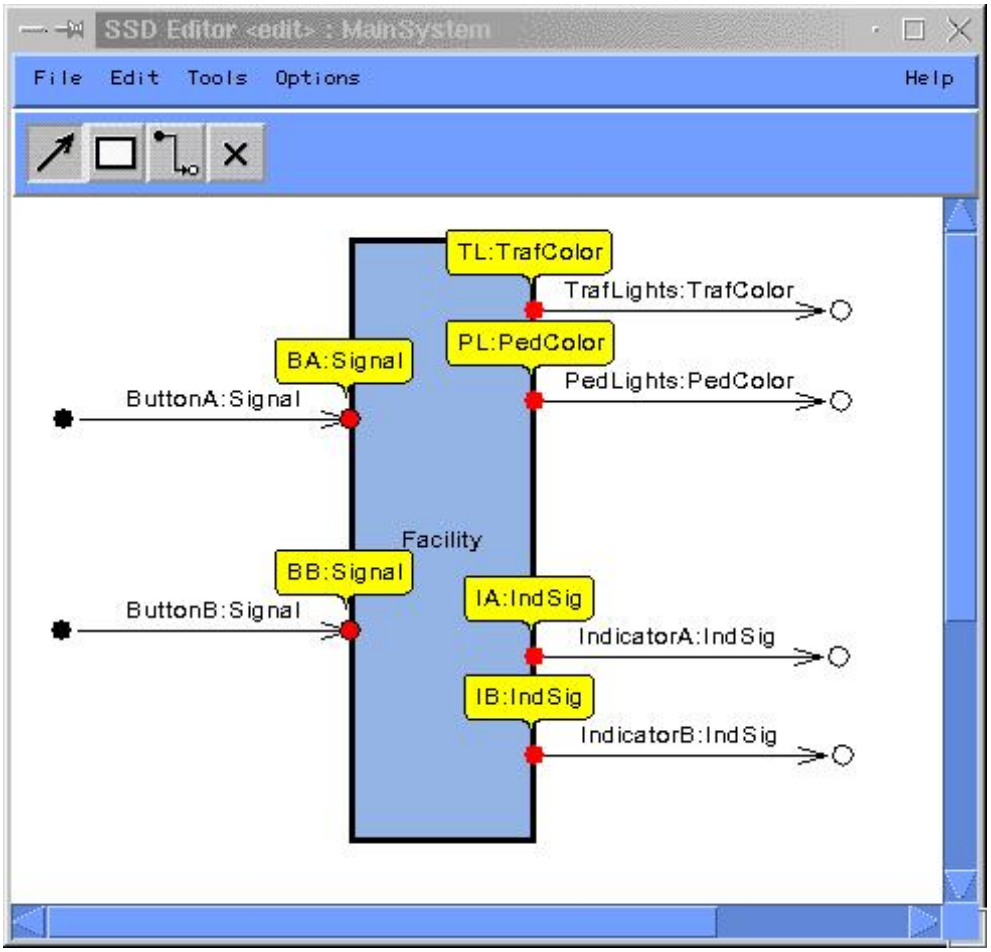

Figure 3.18: ”Facility” component after the ports have been renamed

1. Select the ”Facility” component
2. ”Edit” menu - select ”Substructure” (or double-click `Facility`)
3. A new SSD editor called ”Facility” is displayed. It contains the ports of the ”Facility” component and shows component’s the inside view.
4. Activate the ”Select” button. If necessary, relocate the ports such that the input ports are situated on the left and the output ports on the right, with sufficient space for two components between the ports.

Figure 3.19: SSD editor with the inside view of the ”Facility” component

5. Activate the ”Component” button and insert two new components, ”Merge” on the left and ”Controller” on the right. For the meaning of these new components, refer to ”Structural View of the Traffic Lights Example.”.
6. Drawing in the channels: activate the ”Channel” button, click the input port on the top left, hold the mouse key down, and drag the cursor horizontally into the ”Merge” component. A new dialog field is displayed, asking you to enter the name (*BA*) and the type (*Signal*). In the same manner, enter the other channels as shown in the figure below:
7. If necessary, rename the ports, e.g. ”Request” as ”Req”:

Now the structural view of the traffic lights controller is fully described.

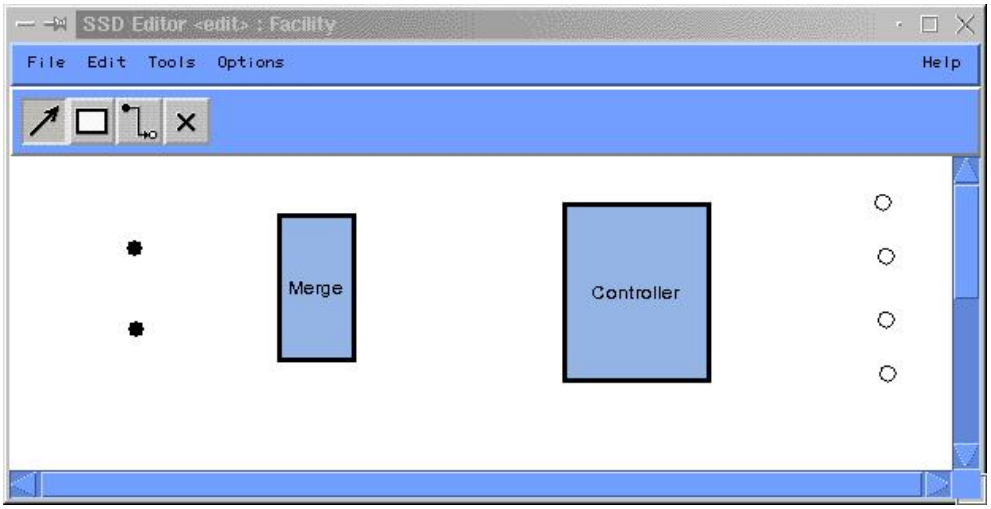

Figure 3.20: The two new components ”Merge” and ”Controller”

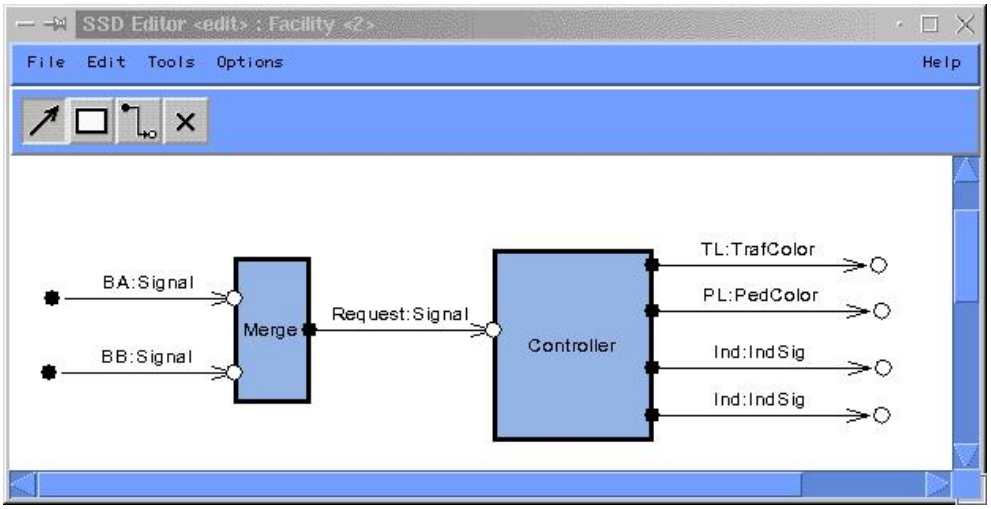

Figure 3.21: ”Merge” and ”Controller” components with new channels

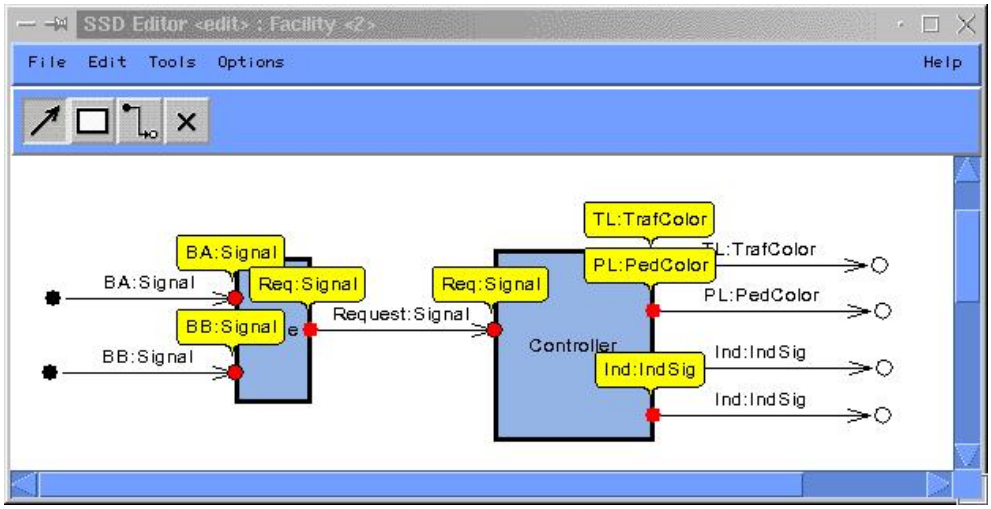

Figure 3.22: ”Merge” and ”Controller” components with renamed ports

Chapter 4

Data View

Diagrams also always contain textual annotations of some kind. To make machine processing of the diagrams possible, these annotations must be written in a formal language. AutoFOCUS uses typed languages. Typed means that a type must be defined for every quantity used. This applies, for instance, to every channel name and every port name. There are also formulas (terms) and patterns (terms with parameters) and they also have types, which, however, follow from the context in most cases. To be able to model systems from the data view, we have to acquaint ourselves with the correct notation of types and terms.

4.1 The AutoFOCUS Type System

AutoFOCUS supports the following two approaches:

- Java: a partial set of the basic types of the Java programming language can be used for typing. In this case, it is not possible to use self-defined types.
- QuestF: QuestF is a functional programming language which combines concepts of Gofer and elements of algebraic specification languages. The basic types of the QuestF language, as well as types which can be introduced with the data construct, can be used for typing.

For each project it must be determined whether Java or QuestF is to be used. It is not possible to mix the two approaches in one and the same project. The models can be simulated in either case, regardless of whether Java or QuestF is used.

4.2 Java

Essentially, Java is used to model projects which are very close to hardware or implementation. As a rule, the signals have to be coded as combinations of the data types boolean (truth values), int (integers), and float (floating point numbers). This makes the models very technical and often difficult to comprehend. Advantages of Java modelings are that a more efficient code is produced and that it is possible to integrate any Java code desired. However, as will be shown later on, this severely restricts the possibilities of model validation (proof of certain properties).

4.3 The QuestF Language

We do not intend to program in QuestF. The language is only used to construct types, to type quantities in the diagrams, to declare constants, and to formalize restrictions.

4.4 Types and Terms (QuestF)

In computer systems, all sorts of information are represented in coded form and processed. To ensure that the representation is interpreted correctly during processing, it must be defined for each type which kind of information it is. In nearly all modeling environments, certain frequently used data types are already defined. In addition, it is possible to introduce user-defined data types. In the following, we will describe how to do this in AutoFOCUS.

4.4.1 Predefined Types

Predefined data types are the basis of the data models and of self-defined data types. In QuestF there are the following three predefined, elementary types:

- truth values: `Bool`
- integers: `Int`
- floating point numbers: `Float`

For reasons of compatibility with the various programming languages and notations, several synonymous names were set for the same types. The use of the synonyms, however, has no influence on functionality.

- truth values: `Bool, boolean, Boolean`
- integers: `Int, int`
- floating point numbers: `Float, float`

4.4.2 Truth Values: Bool

The constant values `True` and `False` and the following operators are available:

Name	Operator	Signature	Remark
equality	`==`	`Bool , Bool -> Bool`	
inequality	`!=`	`Bool , Bool -> Bool`	
negation	`not`	`Bool -> Bool`	
implication	`=>`	`Bool , Bool -> Bool`	
equivalence	`<=>`	`Bool , Bool -> Bool`	
disjunktion	`||`	`Bool , Bool -> Bool`	logical ”or”
conjunktion	`&&`	`Bool , Bool -> Bool`	logical ”and”

With these and with parameters of type `Bool` we can formalize expressions (terms). We will deal with terms in more detail later on.

Example:

If `A,B,C` are boolean parameters, `True => False A || B (A && B) <=> C` are correct terms of type Bool.

4.4.3 Integers: Int

The `Int` data type represents the set of integers in decimal notation. Only in the context of simulation and code generation is the range of values of `Int` restricted to `[-2147483648..2147483647]` and `[-2^31..2^31-1]` (representation of a 32-bit number in two’s complement) respectively. The table below shows the predefined operators for `Int`:

Name	Operator	Signature	Remark
equality	`==`	`Int , Int -> Bool`	=also possible
inequality	`!=`	`Int , Int -> Bool`	
greater	`>`	`Int , Int -> Bool`	
greater or equal	`>=`	`Int , Int -> Bool`	
less	`<`	`Int , Int -> Bool`	
less or equal	`<=`	`Int , Int -> Bool`	
addition	`+`	`Int , Int -> Int`	
subtraction	`-`	`Int , Int -> Int`	
multiplication	`*`	`Int , Int -> Int`	
division	`/`	`Int , Int -> Int`	integer division
remainder	`%`	`Int , Int -> Int`	integer division remainder

Remark: The division (`/`) is an integer division, i.e. the decimal places of the result are cut off.

At present there is no single-digit operator (`- : Int -> Int`) for the negation. Therefore, `-1` must be represented as `0-1`.

Example:

With the operations of the `Int` data type, terms of type `Int` and boolean expressions can be defined:

Int expressions (N, M are parameters of type `Int`):

- `8*25`
- `(0-1)*N`
- `N/M`

Boolean expressions:

- `N > 5`
- `N*M == 26`
- `(0-15)*M < N`

4.4.4 Floating Point Numbers: Float

In QuestF, floating point numbers are called `Float`. Floating point numbers (e.g. 0.0, 5.38, 3e10) are given in the same syntax as in C or Java. Numbers of type `Int`

are distinguished from numbers of type `Float` by ".0". In QuestF, the following operators are available for the `Float` type:

Name	Operator	Signature	Remark
equality	`==`	`Float , Float -> Bool`	not suitable
inequality	`!=`	`Float , Float -> Bool`	not suitable
greater	`>`	`Float , Float -> Bool`	
greater or equal	`>=`	`Float , Float -> Bool`	
less	`<`	`Float , Float -> Bool`	
less or equal	`<=`	`Float , Float -> Bool`	
addition	`+`	`Float , Float -> Float`	
subtraction	`-`	`Float , Float -> Float`	
multiplication	`*`	`Float , Float -> Float`	
division	`/`	`Float , Float -> Float`	floating point division

Remark: If `==` and `!=` are used, differences between the mathematical equality (or inequality) and the calculated truth value may occur due to calculation inaccuracies that cannot be avoided. Therefore, the operators `==` or `!=` should not be used for floating point numbers.

If one of the two-digit operators is used with one `Int` and one `Float` argument each, the `Int` representation is automatically transformed into `Float` (so-called type-casting). For example, 5.0 * 3 turns into 5.0 * 3.0.

As in `Int`, there is no one-digit operator for the negation in `Float`

4.4.5 Terms

In the following section, the structure and usage of terms in QuestF is described in greater detail. Terms are frequently used, e.g. to define functions and constants and also for the conditions on transitions (see "Behavioral View").

Construction of Terms

In QuestF, terms may consist of the following elements:

- constants and parameters, e.g. `True`, `2` or `X`
- operations, as in `2+`, `True \&\& False` and `5*3.5`

- parentheses, as in (2+3)*5
- conditional expressions such as
  `if x then y else z fi`

  where `x` must be a term of type `Bool` and `y` and `z` can be chosen from any desired type but must all be of the same type.
- functions like `Withdraw(2387,50)`

A syntax description for terms in BNF notation is available here (see URL **??**). Priority rules such as "* binds stronger than +" and "&& binds stronger than ||" are laid down by convention. Examples:

Evaluation of Terms

The evaluation of terms is based on an "eager semantics", i.e. before a function or operation is applied, all arguments of the function are calculated. It is irrelevant whether the values are or are not required for calculating the whole term. The only exception is the `if b then t1 else t2 fi` construct. Here, b is evaluated first, and then, depending on the result (`True` or `False`) the value of the corresponding term `t1` or `t2` calculated.

Example:
The term `(1+1==3) && (10==5/0)` cannot be evaluated in an "eager semantics" because the result of `5/0` is not defined. If the term is represented as `if (1+1==3) then 10==5/0 else False fi`, however, `5/0` need not be evaluated and the whole term can be interpreted as `False`.

4.4.6 Constants

Constant definitions consist of a name (identifier) and a value. Once defined, constants can be used repeatedly in the following, but their values cannot be changed.

A constant definition has the following form:

```
const <constIdentifier> = <value>;
```

with `constIdentifier` being the constant's identifier and `value` its value. The type of the new identifier is determined automatically by means of type inference (determination of the most general type).

Example:

The following definition generates a new constant `TRed` of type `Int` with the value `5`:

```
const TRed = 5;
```

4.4.7 Type Definitions

In the section dealing with the structural view, we have chosen identifiers for data types which were (to a great extent) freely selectable. The types predefined in QuestF are only partially suited to special problems. In particular, it is important that enumeration types, tuples, and variants are available for modeling as well. For example, the modeling of the traffic lights controller uses its own data type `TrafColor` to represent the colors of the traffic lights. If implemented correctly, interpreting an integer for the traffic lights display is no problem for a computer system. Humans, however, find it easier to deal with meaningful identifiers. Furthermore, type definition is a prerequisite for the problem-specific control of inputs. QuestF knows the data construct for type definition.

Data Definitions

A data type is the combination of a set of values and operations. The set of values (also called sort) consists of all elements a quantity (e.g. a channel or a parameter) may adopt. The operations are required for defining conditions and calculating new values. With the aid of the `data` construct, new sorts with certain standard operators (constructor functions, selector functions, and discriminator functions) can be defined in QuestF. In the following, such a definition is also called data definition.

The `data` construct in QuestF has the following syntax:

data *newDataDef* = *Constr1* | *Constr2* | ... | *ConstrN*;

newDataDef is the identifier of the newly defined type and *Constr1* to *ConstrN* are identifiers for new constructor functions (see Section 4.4.7). With the aid of the constructor functions, the terms of the newly defined sort are constructed. The constructors are separated from each other by vertical strokes (`|`).

Enumeration Types

If all constructor functions of a data definition are zero-digit, i.e. if they are constants, these constants are the only terms of the new type. The definitions

Example:

```
data TrafColor = Red | RedYellow | Green | Yellow;
data PedColor = Walk | Stop;
```

set a type `TrafColor` with four different constants (`Red`, `RedYellow`, `Green` and `Yellow`) and a type `PedColor` with the two constants `Walk` and `Stop`.

Constructors may also have parameters. With these parameters new data definitions can be composed from other predefined or self-defined types.

Variants

Example:

```
data ExtInt = I2E(E2I:Int) | Undef;
data Number = I2N(N2I:Int) | F2N(N2F:Float);
```

Variants are a union of several types so to speak. A term of type `ExtInt` has as value either one that is analogous to the `Int` type or the `Undef` constant. Terms of type `ExtInt` are, for example, `I2E(0)`, `I2E(5)`, and `Undef`. The `I2E` constructor (”integer to extended integer”) generates a term of type `ExtInt` from a term of type `Int`. However, the operations defined for the `Int` type are not applicable to terms of type `ExtInt`. Analogously, `I2N` is the union of `Int` and `Float`. Enumeration types are special variants which have only zero-digit constructors.

Tuples

Example:

```
data Datum = TMJ(tag:Int, monat:Int, jahr:Int);
```

The above data definition generates a new type named `Datum`. In our example, a constructor `TMJ` (tag/monat/jahr) is defined which integrates three terms of type `Int` into a triple, with the components not necessarily having to be of the same type.

`tag`, `monat` and `jahr` are identifiers for selector functions (see Section 4.4.7) (partial functions for accessing the arguments of a constructor, see below). If no selector functions (or "selectors" for short) are given, predefined selectors are generated automatically.

The precise syntax of the data definitions may be gathered from the syntax description in BNF notation (**??**).

Example:
A tuple for the displays of the lights for cars and the lights for pedestrians:
`data TrafLights = Lights(TrafColor,PedColor);`

Recursive types such as lists and trees can also be defined in QuestF. "Recursive" means that during type definition one already makes use of the type to be newly defined.

Example:

```
data IntList = Cons(tail:IntList,head:Int) |
               EmptyList;
data IntTree = Node(left:IntTree,right:IntTree,
                    element:Int) |
               EmptyIntTree;
```

Identifiers

In data definitions, new identifiers for types, constructors, and selectors have to be given. To ensure that the operations are unambiguous, the same identifiers may not be used in the data definitions of a project more than once. This means that, for example, `Red` may not be used simultaneously as constructor for `PedColor` and `TrafColor`. In addition, identifiers invariably have to begin with an alphabetic character and may not be identical with any key words or identifiers used in predefined data types (Int, Bool, 0, 1, 2.4, True, ...).

Constructor Functions

Each data definition comprises the definition of one or several constructors for constructing the elements of the type to be newly defined. In the `TrafColor` example four constructors are defined: `Red`, `RedYellow`, `Green`, and `Yellow`. Having no parameters, these constructors may also be regarded as constants (i.e. zero-digit functions). The `data` definition of `TrafColor` thus defines the following four constructor constants (zero-digit constructor functions):

- `Red       : TrafColor`
- `RedYellow : TrafColor`
- `Green     : TrafColor`
- `Yellow    : TrafColor`

If the constructors have one or several parameters, corresponding single- or multi-digit constructor functions are defined. The data definition of `Datum` defines the following constructor function:

- `TMJ : Int , Int , Int -> Datum`

Discriminator Functions

Further, for every constructor function a discriminator function is generated automatically in QuestF, which allows us to determine whether a term was or was not generated with the respective constructor function. For instance, the following discriminator functions are generated in the `TrafColor` example:

- `is_Red : TrafColor -> Bool`
- `is_Yellow : TrafColor -> Bool`
- `is_RedYellow : TrafColor -> Bool`
- `is_Green : TrafColor -> Bool`

If, for example, the term `is_Red(t)` is evaluated, it yields `True` if `t` is evaluated as `Red` and `False` in all other cases. By defining `RatZahl`, three discriminator functions are generated:

- `is_Ganz : RatZahl -> Bool`
- `is_Rein : RatZahl -> Bool`
- `is_Gemischt : RatZahl -> Bool`

A call `is_Ganz(t)` yields the value `True` if `t` is a term of the form `Ganz(n)` and `False` if `t` is a term of the form `Rein(n,m)` or `Gemischt(n,m,o)`, with arbitrary partial terms n,m,o of type Int. The `is_Ganz` and `is_Rein` discriminator functions behave analogously. This means that the discriminator function makes distinctions according to the main operator of the term.

Selector Functions

For constructors with parameters, a selector function is defined for each parameter. Selector functions (or "selectors" for short) allow us to calculate the original parameter the term was constructed with. The data definition for `Datum` comprises the following three selector functions:

- `tag : Datum -> Int`
- `monat : Datum -> Int`
- `jahr : Datum -> Int`

Selectors are partial functions. Partial functions are not defined for all sorts of input. For instance, the `tag` function is only defined for values generated with the `TMJ` constructor function. Otherwise the function application does not yield a defined result. For this reason, we must make sure in advance that the value to be selected was indeed generated with the corresponding constructor. This can be realized by means of the discriminator functions.

Example:
(Calculation of the `Float` value of a rational number (RatZahl)):

```
if is_Ganz(r) then
  ival(r):Float
else
  if is_Rein(r) then
    r_zaehler(r):Float / r_nenner(r):Float
  else
    ganzzahl(r):Float + (g_zaehler(r):Float /
                         g_nenner(r):Float)
  fi
fi
```

Remark: `:Float` causes the `Int` value to be converted into a `Float` value. This is necessary for a floating point division to be used.

Standard Operators

Upon each definition of a type *T*, equality and inequality are automatically defined as standard operators:

Name	Operator	Signature	Remark
equality	`==`	*T* , *T*`-> Bool`	
inequality	`!=`	*T* , *T*`-> Bool`	

Summary: Every data definition introduces a new type and, in addition, certain standard functions with canonical properties. Altogether, the result is a so-called *data type*.

Patterns

Patterns are terms which consist solely of parameters and constructor functions. Other operations or functions are not admissible in patterns. Pattern matching allows us to find out whether arbitrary terms satisfy a pattern. To accomplish this, the term must correspond to the pattern, except for the parameters in the pattern. If the pattern contains any free parameters (randomly chosen identifiers to which no value has been assigned), they are bound to the corresponding terms during ”matching”.

Example:

Value: `TMJ(28,2,1999)`

Pattern: `TMJ(Tag,Monat,2000)`

In this case, matching fails because the years are different (1999 != 2000).

Example:

Value: `Withdraw(28,2,1999)`

Pattern: `Withdraw(Tag,Monat,1999)`

Here the given value can be matched with the pattern by binding `Tag` to `28` and `Monat` to `2`

Pattern matching is particularly important for input patterns on transitions (see also ”Behavioral View”).

4.4.8 Example of Use

Type Definitions for the Traffic Lights Controller

For the traffic lights controller 3 modules (DTDs) are created for type definitions:

- `Signal` for the signal request
- `Lights` for the traffic light displays
- `TimeConstants` for the time behavior (switching speed) of the traffic lights

DTD: Signal

In the `Signal` module the type definition stands for the signal request. Terms of type `Signal` may adopt the value `Present` at most. If no signal is present, this is implicitly expressed by the fact that no message is sent. Therefore, the `Signal` type has only one constructor, namely `Present`:

```
data Signal = Present;
```

DTD: Lights

The `Lights` module defines the enumeration types for the displays of the lights for cars and the lights for pedestrians. The lights for cars can display the four color combinations red, red/yellow, green, and yellow. For this purpose, the type `TrafColor` is set with the following data definition:

```
 data TrafColor = Red
| RedYellow
| Green
| Yellow;
```

The lights for pedestrians do not only feature the color signals green (`Walk`) and red (`Stop`), but also the optical display for the signal request (`On` and `Off`). For this purpose, the following two enumeration types are defined:

```
  data PedColor = Walk
 | Stop;

  data IndSig = On | Off;
```

DTD: TimingConstants

In the `TimingConstants` module, the values for the timer variable`T` are set, which determines how long the traffic lights shall remain in a particular state. This may be accomplished by means of constants. If the traffic lights' time behavior is to be changed, the constants are simply set to other values. For the traffic lights controller, the following three constants are defined which determine the time behavior:

```
  const TGreen = 10;

  const TYellow = 2;

  const TRed = 5;
```

4.5 Instructions for QuestF

In the following, we will explain step by step how the required type definitions for our traffic lights controller can be created in AutoFOCUS. For more details about the type definitions in AutoFOCUS, see:

- Predefined Types
- Terms
- Constants
- Type Definitions
- Type Definitions for the Traffic Lights Controller

4.5.1 Creating a New DTD Document

Start AutoFOCUS. The window of the project browser with the ”Pedestrian Traffic Lights” project is displayed (whose creation was dealt with in ”Instructions: SSDs”).

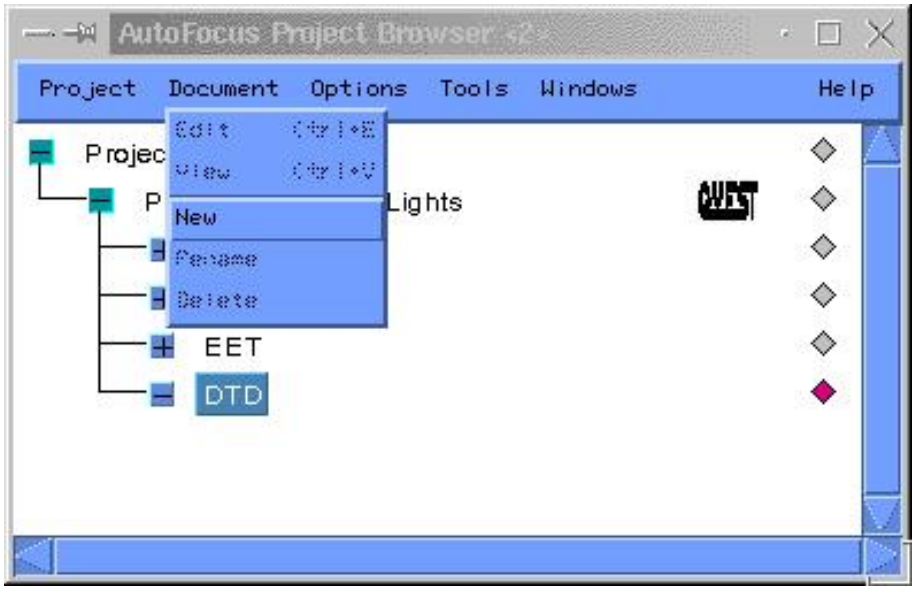

Figure 4.1: Project browser with the”Pedestrian Traffic Lights”project

Expand the list of the various views of the ”Pedestrian Traffic Lights” project by single-clicking on the + symbol and select the ”DTD” entry. In AutoFOCUS, data type definitions are defined and filed in DTD documents. To create a new DTD document, select ”New” from the ”Document” menu.

Enter *Lights* as the new DTD document’s name and confirm by clicking ”Ok”.

4.5.2 The DTD Editor

The window of the DTD editor opens. If you wish to make any corrections later on, the project browser offers the possibility of expanding the ”DTD” entry. On doing so, the ”Lights” entry is displayed below ”DTD”. By double-clicking on ”Lights” you can at any time start the DTD editor with the data definitions of ”Lights” in order to make further corrections.

The window of the DTD editor consists of a menu bar, a text field showing the name of the open DTD document, and a ”DTD Description” text window in which

Figure 4.2: ”New DTD” dialog box for the ”Lights” DTD

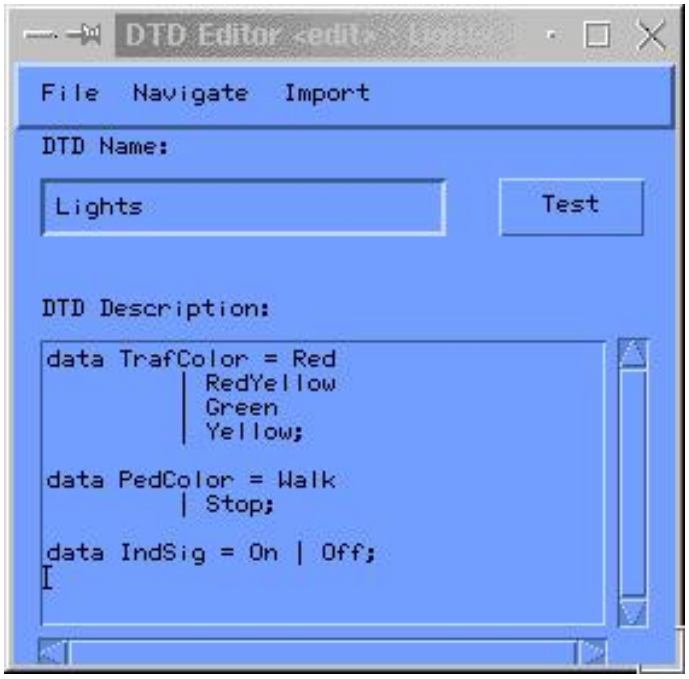

Figure 4.3: The DTD editor with the ”Lights” DTD

the data type definitions are to be entered in the language QuestF. This is done in the manner customary for text editors. By means of the ”Test” button, the data type definitions entered can be checked for syntactic correctness. In ”DTD description”, any number of data types, constants (and functions) can be defined.

Take the entries in the ”DTD Description” text field from the above screenshot. These are the three enumeration types ”TrafColor” (the colors of the lights for cars), ”PedColor” (the colors of the lights for pedestrians), and ”IndSig” (the two signals defined for the indicator lights). No function definitions are required for the traffic lights example.

Figure 4.4: Result of the syntax check of the entries in the DTD document

Test your entries by clicking on the ”Test” button. If any errors are reported, correct them. However, don’t be confused by the window title ”Error occured!”. The message bears this title even if the DTD has been parsed successfully. Save the DTD document with ”Save & Close” from the ”File” menu. The new ”Lights” document is listed below the ”DTD” entry in the project browser. In the same manner you create the two other DTD documents required for the traffic lights example.

In the ”TimeConstants” document, the time constants are set which determine the duration of the various traffic light phases. The ”Signal” data type defined in the ”Signal” document is an enumeration type which consists solely of one (zero-digit) constructor. A signal received from one of the two traffic light buttons is indicated by the value ”Present”. If no button has been pushed this is expressed by channels without values. The values of channels are not permanent during simulation. They are deleted or cleared after one time step in each case.

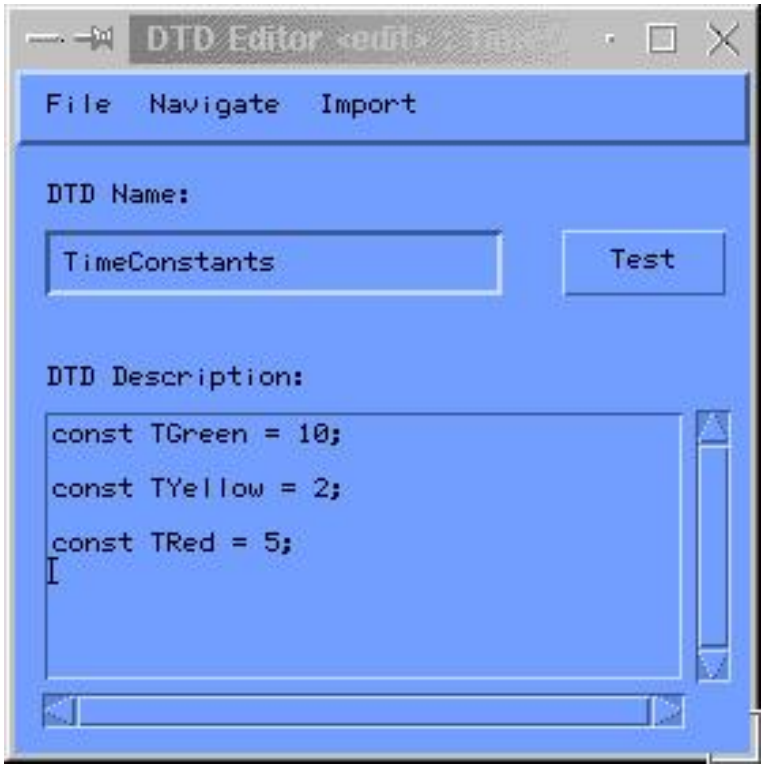

Figure 4.5: ”TimeConstants” DTD document with the definition of the time constants

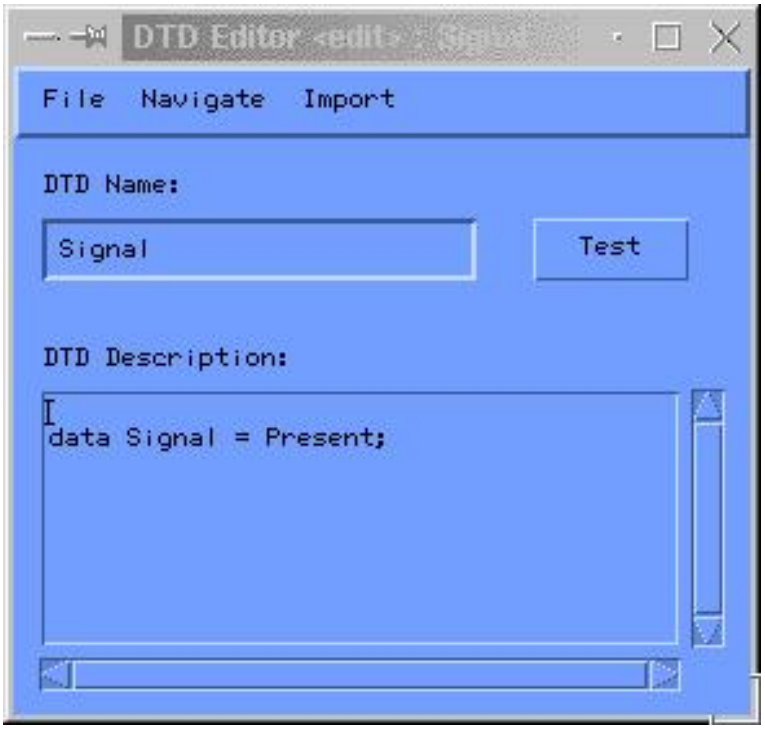

Figure 4.6: ”Signal” DTD document with the definition of the ”Signal” data type

In general, it is irrelevant into how many DTD documents the data type definitions for an AutoFOCUS model are split up. All definitions are available in the entire model. If the data type definitions are split up into several documents, this is only done to achieve a meaningful subdivision into related parts.

Chapter 5

Interaction View

5.1 The Communication Behavior

The system structure describes the static view of a distributed system. It is called *static*, because it does not change during the runtime of the system. What changes during a system run is the signals on the channels. The behavioral view of the system describes how the components react to these signals. Giving a detailed and complete description of the behavior of individual components is a difficult task which can usually not be realized in one single step. For this reason it makes sense to start by describing important scenarios of interactions between the components in an exemplary fashion and to provide a complete description of the component's behavior only later on. *Interaction scenarios* serve to lay down the time sequence of certain messages. In the traffic lights controller example, one scenario would look as follows: after a pedestrian has pushed the button, the indicator lamp lights up to acknowledge the signal request. Interaction scenarios can also describe more complex interactions in which several components are involved. As a rule, such scenarios are described by means of sequence diagrams. There are many different kinds of sequence diagrams which use more or less the same concepts, differing from each other only in details. Common standards are MSCs (message sequence charts) and sequence diagrams in UML (Unified Modeling Language). AutoFOCUS provides sequence diagrams (extended event traces, EETs) for the description of the interaction view.

Sequence diagrams have various fields of application and can be used for different purposes in different phases of the software development process:

- In early stages, the basic functionality of the system or its behavior in error situations can be specified in an exemplary fashion. In our modeling of the traffic lights controller, EETs are used to describe elementary interactions.

- Later in the development process, system descriptions provided by other views of the system can be checked against EETs to find out whether they fulfill the functionality specified in them.
- During the validation of a system design, EETs can be used to log simulation runs.

5.2 Sequence Diagrams (Extended Event Traces, EETs)

Extended event traces (EETs) are used to provide exemplary descriptions of the interaction of several components, either among each other or with their environment.

Usually, EETs are assigned to (associated with) a part of the system structure. AutoFOCUS provides for two forms of association:

1. association with a single component
2. association with a system structure diagram (SSD)

In the first case, a typical input/output sequence of the associated component is represented. Here, only the input/output ports of the component should be used in the sequence diagram.

In the second case, the sequence diagram represents a typical interaction between the components of the system structure diagram and its environment. Here, only channels, ports, and components should be used in the sequence diagram which exist in the associated system structure diagram. Sequence diagrams focus on an exemplary representation of the possible interactions or the input/output behavior. For this reason, the interaction patterns need not be complete. The figure below shows, in two EETs, the signal request of the traffic lights controller which relates to the diagrams in Figure 5.2 and Figure 5.3 in the section System Structure Diagrams:

The individual elements of Figure 5.1 will be explained in more detail in the following.

5.2.1 Actors (Axes)

The vertical axes in an EET diagram represent time axes which correspond to components in system structure diagrams. Just as in system structure diagrams,

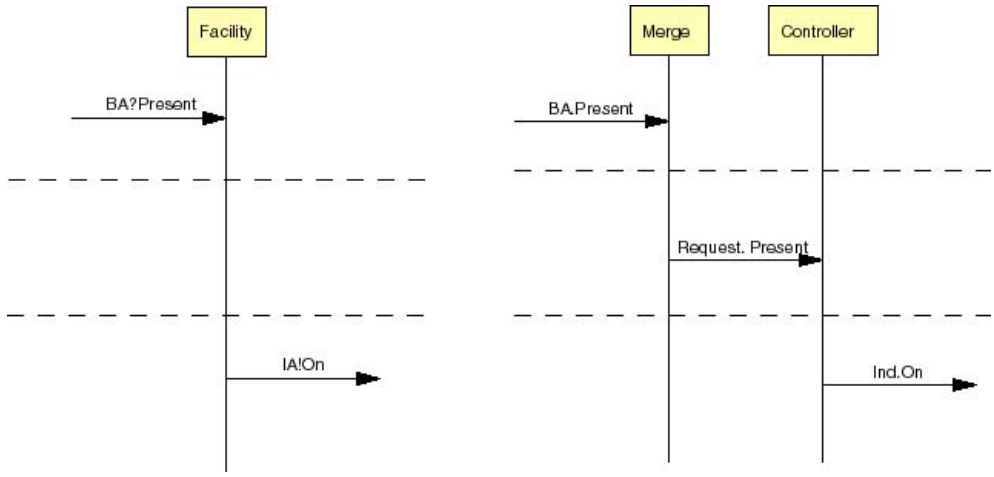

Figure 5.1: Figure 1: Component-related (1.) and structure-related (2.) EET

the environment is not modeled as a separate component, but is implicitly contained in every EET document (see also section ”Messages”). Figure 5.2 shows the components of the set of traffic lights (`Merge` und `Controller`).

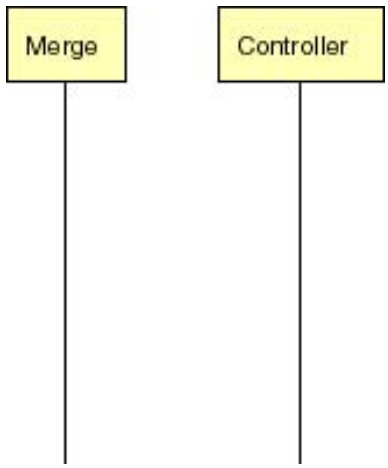

Figure 5.2: Figure 2: Component axes of the set of traffic lights

5.2.2 Messages

Messages are represented by horizontal arrows between different axes or components. The direction of the arrows indicates the direction of the message flow.

A message sent from the `Merge` to the `Controller` component is symbolized by an arrow from `Merge` axis to the `Controller` axis. Messages from the environment are represented by arrows from the outside right or the outside left. Analogously, messages to the environment are represented by arrows that point to the outside right or the outside left.

Message arrows can be annotated with character strings indicating the content of the message as well as the channel or port through which the message is sent. The representation used for channels differs from the one used for ports:

- Value on channel: *channel* . *Value*
- Value on input port: *Input Port* ? *Value*
- Value on output port: *Output Port* ! *Value*

The question mark (`?`) indicates that a value is read from an input port, and the exclamation mark (`!`) that a value is written to an output port. To achieve consistency of the interaction view (EET) and the structural view (SSD), the channel or port used for the message must also be contained in the system structure diagram, and must have the same source and target components. The *value* is a randomly chosen term, which, however, must have the same type as the associated channel or port.

Figure 5.3 shows a signal request relating to the `Facility`component (left) as well as to the refinement, which consists of the `Merge` and `Controller`components (right).As mentioned, only channels of the associated structure diagram should be used in the EET on the right-hand side of Figure 5.3. The EET on the left-hand side, by contrast, relates only to the `Facility` component and its ports. This is why no channel names are used here, but the port names `BA` and `IA`, respectively.

5.2.3 Explicit Time Steps (Ticks)

The order in which the messages in the sequence diagram are represented from top to bottom indicates the time sequence of the events. However, events following each other in the EET need not necessarily happen one after the other, but may also occur simultaneously. In order to explicitly rule out concurrency, symbols for time steps (also called ticks) can be included. They are symbolized by horizontal broken lines. Events separated by a time step definitely occur one after the other, whereas all events between neighboring ticks may happen simultaneously. Figure 5.4 shows the signal request with precise division into time intervals.

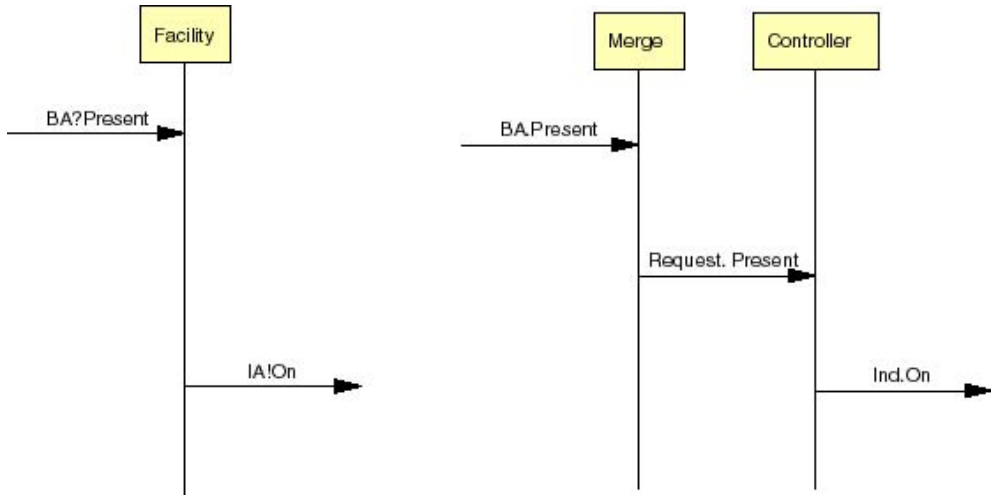

Figure 5.3: Figure 3: Signal request via Button A

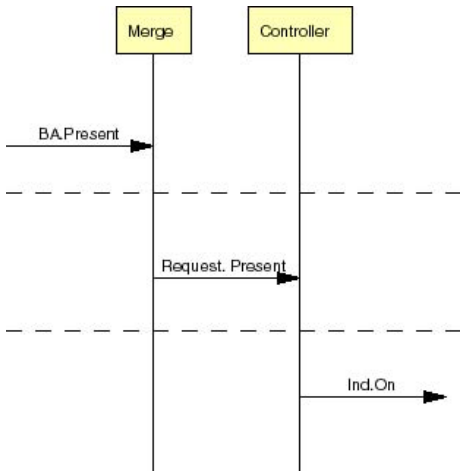

Figure 5.4: Figure 4: Signal request via button A, divided into time intervals

5.2.4 Structuring Elements (Boxes)

As is the case with system structure diagrams, sequence diagrams also offer possibilities of structuring interaction scenarios hierarchically. In EETs, so-called boxes can be used, in which a set of sub-EETs can be named. Since any of the sub-EETs can be chosen as interaction for the period in question, boxes offer the possibility of structuring as well as of introducing behavior variations in EETs.

Example:
If a signal on the lights for pedestrians is requested, it is in many cases irrelevant whether the signal comes from push-button A or push-button B. Therefore, it makes sense here to combine the two interactions ”RequestSignal A” and ”RequestSignal B” (see Figure 5.5) in a box (see Figure 5.6).

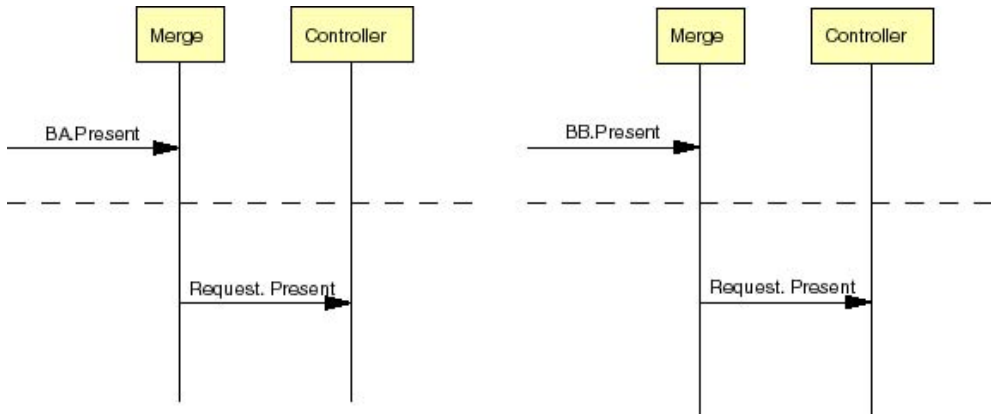

Figure 5.5: Figure 5: Scenarios for signal requests on buttons A and B

5.2.5 Loops (Indicators)

Partial sections of EETs can be provided with so-called *indicators* characterizing optional or repeatable sections. The indicators may be annotated in order to determine the number of runs. The following kinds of repetitions are permissible in EETs:

- `1-*` : at least once
- `0-*` : any number of times (possibly not at all)
- `0-1` : optional

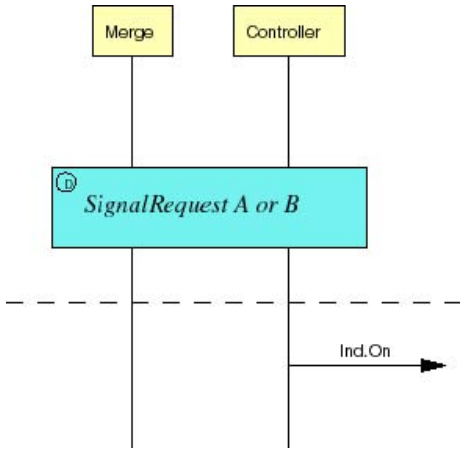

Figure 5.6: Figure 6: Signal request with activation of the indicator light. The box acts as a placeholder for both signal requests.

Example:

Figure 5.7 shows the entire run of the traffic lights. First, the system is initialized. Being irrelevant for the later run, the initialization is described in the `InitializeSystem` box. Then a loop with the normal traffic lights cycle begins: as soon as a `Request` is present, the traffic lights switch, after several intermediate steps, to red for drivers and green for pedestrians and then back to the initial state. This process can repeat itself any number of times. Therefore, the indicator is annotated `0-*`.

Remarks:

- The `Request` box corresponds to the entire signal request as shown in Figure 5.6.
- The port names appearing here belong to the ports of the `Facility`component.

5.2.6 Axis Refinement

Another option for a hierarchical description is axis refinement. Analogously to the decomposition in the associated system structure diagram, it is also possible to fold out an axis of an EET. The sub-document should then, if possible, contain all components of the SSD sub- document associated with it. In order to ensure

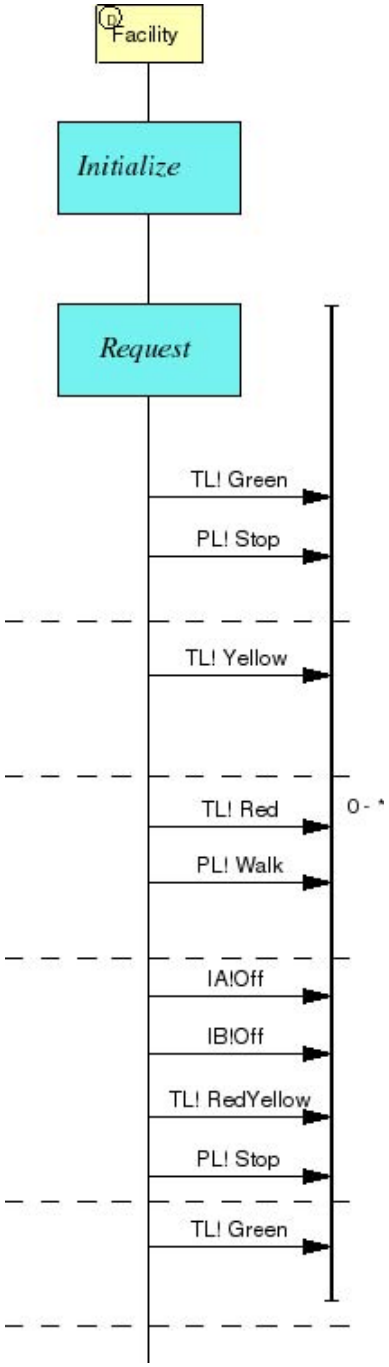

Figure 5.7: Figure 7: Entire run of the traffic lights

consistency, all inputs/outputs of the super-EET must also appear in the refined view. This leads to complex consistency conditions. Therefore, axis refinement is only rarely used and usually does not extend over more than one step.

Example:

Figure 5.8 shows the entire run of the traffic lights relating to the `Facility` component and relating to its decomposition. The (encircled) ”D” on the upper left of the axis label indicates that a refined view exists, in which this axis is divided into the subcomponents of the referenced component from the system structure diagram.

5.3 Instructions for EETs

In the following, we will explain step by step how interaction diagrams (more precisely: extended event traces, EETs) for our traffic lights controller can be created in AutoFOCUS. For more details about EETs, please refer to the following sections:

- Interaction View (see Chapter 5)
- Sequence Diagrams (see Section 5.2)

It is essential to have the system structure diagram (see Section 3.3) available.

5.3.1 Creating a New EET Document

Start AutoFOCUS. The project browser window with the ”Pedestrian Traffic Lights” project is displayed (whose creation was explained in the section ”Tool Operation AutoFOCUS - System Structure Diagrams (see Section 3.3)”). Expand the list of the different views of the ”Pedestrian Traffic Lights” project by single-clicking the + sign and select ”EET.” In AutoFOCUS, interaction diagrams are defined and filed in EET documents. To create a new EET document, select ”New” from the ”Document” menu. A dialog window is displayed and you are asked to enter the new document’s name. Type *Mainsystem* and confirm by clicking ”Ok.” After that, the EET editor opens.

5.3.2 The EET-Editor

The window of the EET editor consists of a menu bar, a tool bar, and the drawing space.The tool bar symbols have the following functions (the respective function

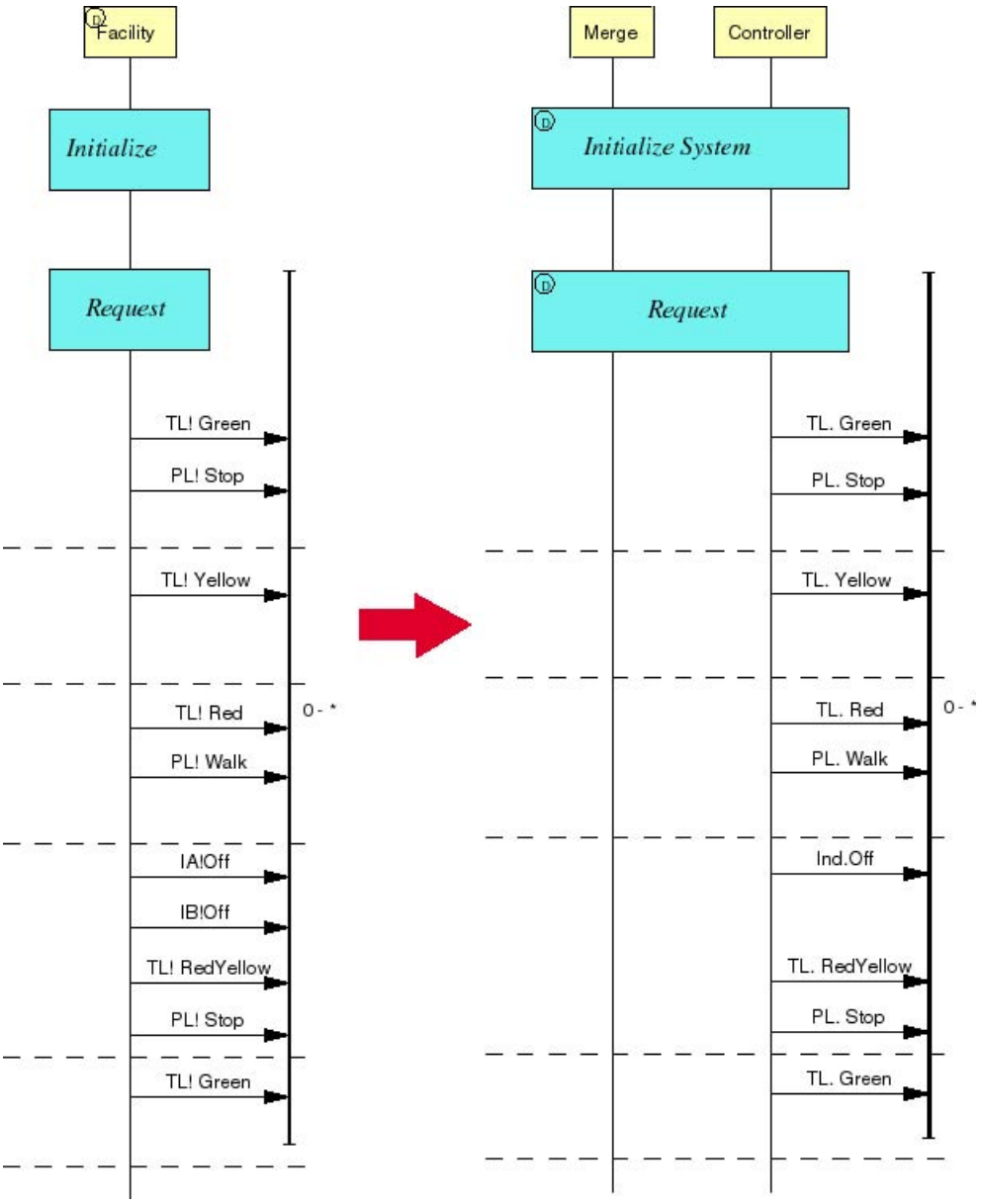

Figure 5.8: Figure 8: Axis refinement

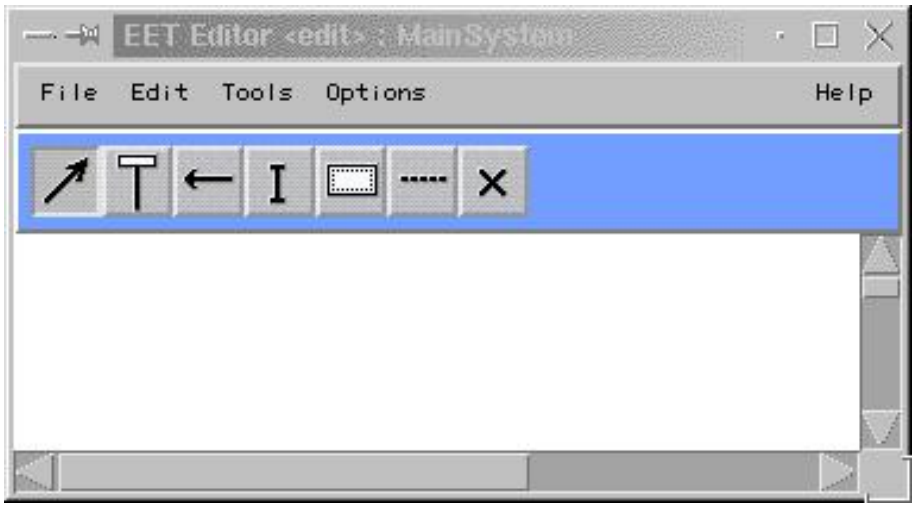

Figure 5.9: The EET editor. The empty EET document ”Mainsystem” is open.

is selected by clicking; precisely one function is active at any point of time):

Symbol	Name	Function
	Select	Selects individual elements of the drawing (highlighting them red)
	Axis	Creates a vertical time axis
	Message	Creates a horizontal arrow for a message
	Indicator	Creates an indicator along a time axis
	Box	Creates a box for structuring
	Tick	Creates a broken line for an explicit time step (tick)
	Delete	Deletes diagram elements by clicking them with the mouse

For general advice on the operation of the editors in AutoFOCUS, please consult

the operating instructions for editors (see Section 7)

5.3.3 Interaction with the Environment

First we describe how the traffic lights controller as an individual component (Mainsystem) interacts with its environment.

5.3.4 Creating Axes

Insert a vertical time axis in the EET editor. Activate the ”Axis” tool symbol and single-click on the drawing plane, at the spot where the axis head with the label is to be located (at the top in the middle). A dialog box is displayed, in which you enter the label *Facility* confirm by clicking ”Ok.”

5.3.5 Creating Boxes

The first part of the interaction we want to include is the initialization phase of the traffic lights controller. It is followed - sooner or later - by a request for a green phase for pedestrians. At this stage of the design, we are not interested in what exactly happens during these two phases. Therefore, we add two boxes for them. Select the box function from the tool bar and click on the axis, a little below the axis head. A dialog box opens, in which you enter the name *Initialize*. The first box is displayed. The second box, *Request* is created in the same fashion.

5.3.6 Creating Messages

Now we are going to specify the messages which the traffic lights controller sends to the environment after a green phase for pedestrians has been requested.First, a signal for ”green” is sent to the lights for cars and a stop signal to the lights for pedestrians. To create these messages, select the ”Message” symbol from the tool bar and use the mouse, beginning at the axis (or slightly to the right of the axis), to drag the cursor horizontally a little bit to the left. After you have released the mouse key, you are requested to give the new message a name. We call it *TL!Green*, in accordance with the port name `TL` which was laid down in the system structure diagram, and the signal value `Green`, defined in the ”Lights” DTD. Create the second message, labeled `PL!Stop`, in the same manner.

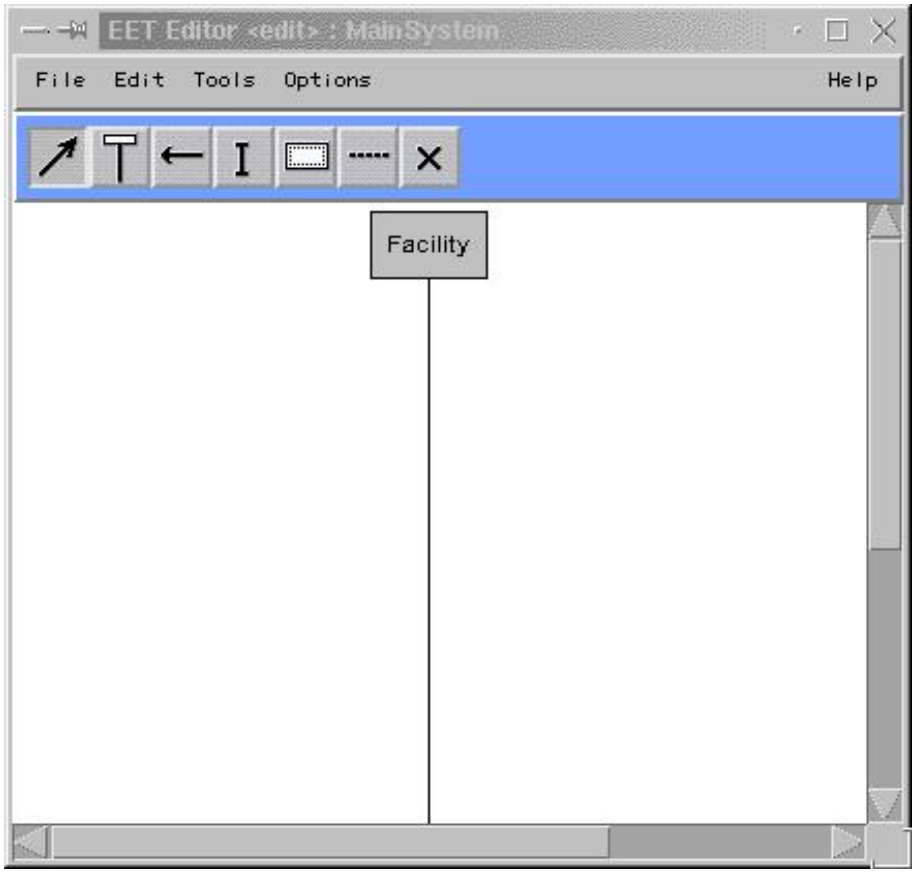
EET Editor <edit> : MainSystem
File
Edit
Tools
Options
Help
Facility

Figure 5.10:

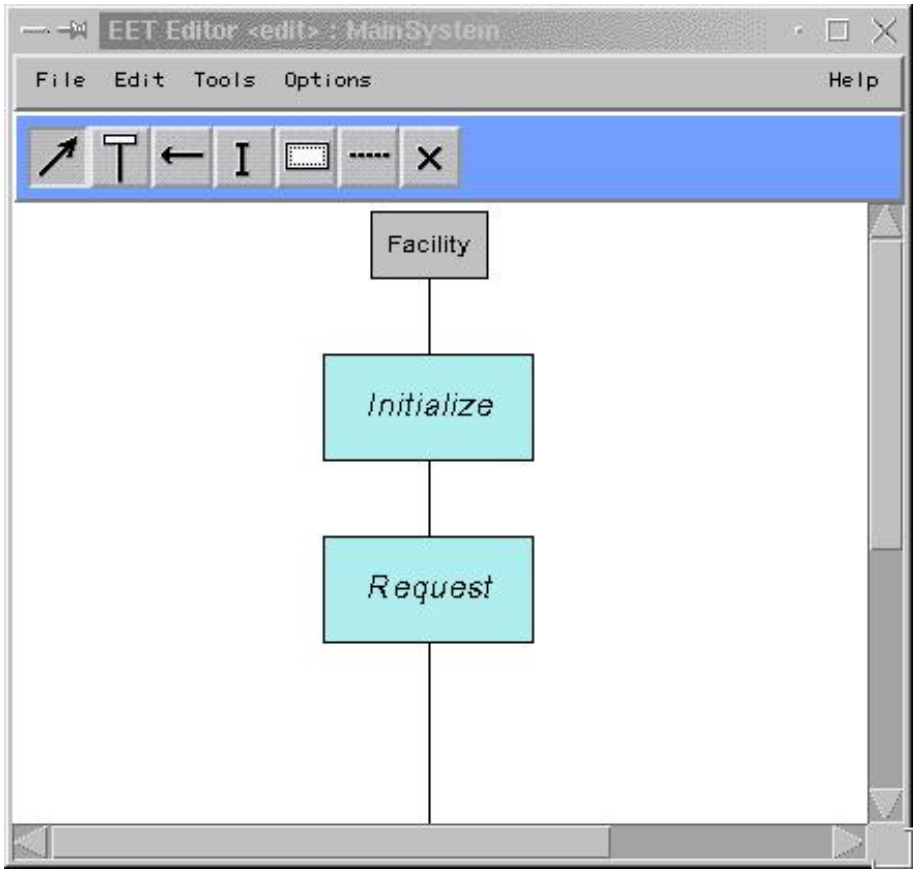

Figure 5.11: EET Editor Box

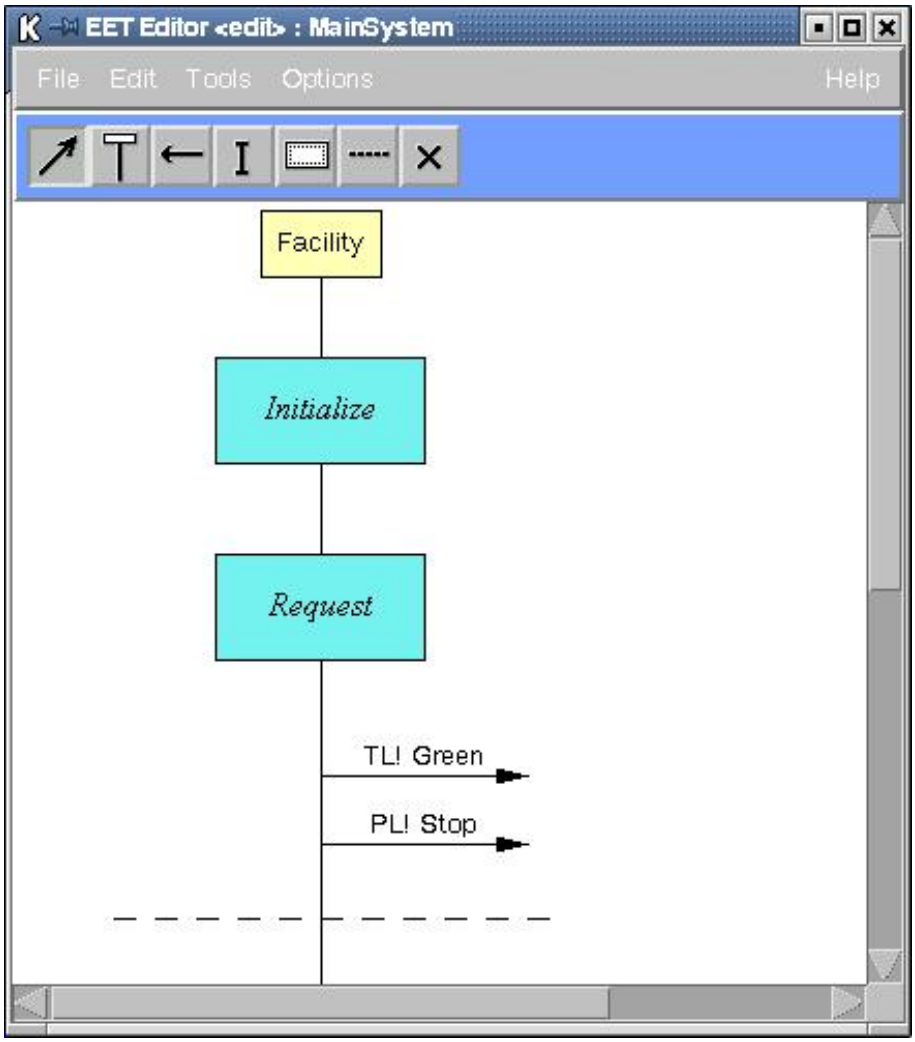

Figure 5.12: The EET editor with two boxes, two messages, and a tick

5.3.7 Inserting Ticks

The first (explicitly drawn) time step is now finished. In AutoFOCUS, we indicate this by inserting a tick. For this purpose, activate the tick function from the tool bar and click on the axis, at the spot where the axis is to intersect the broken line. A tick emerges, as shown in the illustration above.Now complete the diagram such that an entire switching period is shown. For this purpose, take the messages and ticks from the figure below:

5.3.8 Inserting an Indicator

Finally, we want to show that the signal ”Request”, together with the ensuing series of messages, can be repeated any number of times. For this purpose, we add an indicator.Select the indicator function from the tool bar. Now an indicator can be created by clicking on the lowest arrow (lower end of the indicator) and afterwards on the ”Request” box (upper end of the indicator). A window named ”Repetition” is displayed, in which the desired multiplicity can be selected. Select `0 - *`. Now you have created the interaction diagram of the ”Facility” component. Save the document without closing it, using the ”Save” command from the ”File” menu.

5.3.9 Refinement in the Interaction Diagram

Next, we want to carry out a refinement step analogous to the decomposition step in our system structure diagram. This enables us to specify the interaction between the `Merge` and `Controller` subcomponents of Facility. To decompose the `Facility`-axis, we select the axis by means of the selection tool and choose the ”Sub-Structure” command from the ”Edit” menu. After that, a new EET editor window emerges.Now insert two axes labeled `Controller` and `Merge` Instead of making the same entries as in the EET of `Facility` all over again, the obvious thing to do now is to use Copy & Paste. For this purpose, select the two boxes, the messages, and the ticks from the EET of `Facility` with the STRG key held down. Now select the ”Copy” command from the ”Edit” menu. Switch to the new EET window and carry out ”Paste” from the ”Edit” menu. Now your new EET should look like the one shown in the figure above.An additional form of structuring is achieved by creating separate EETs for the two boxes. For this purpose, we select the ”Initialize” box and choose the ”Sub-Structure” command from the ”Edit” menu. In this select window, select ”New” and draw the following EET in the new editor window: The `Request` box is decomposed in the same

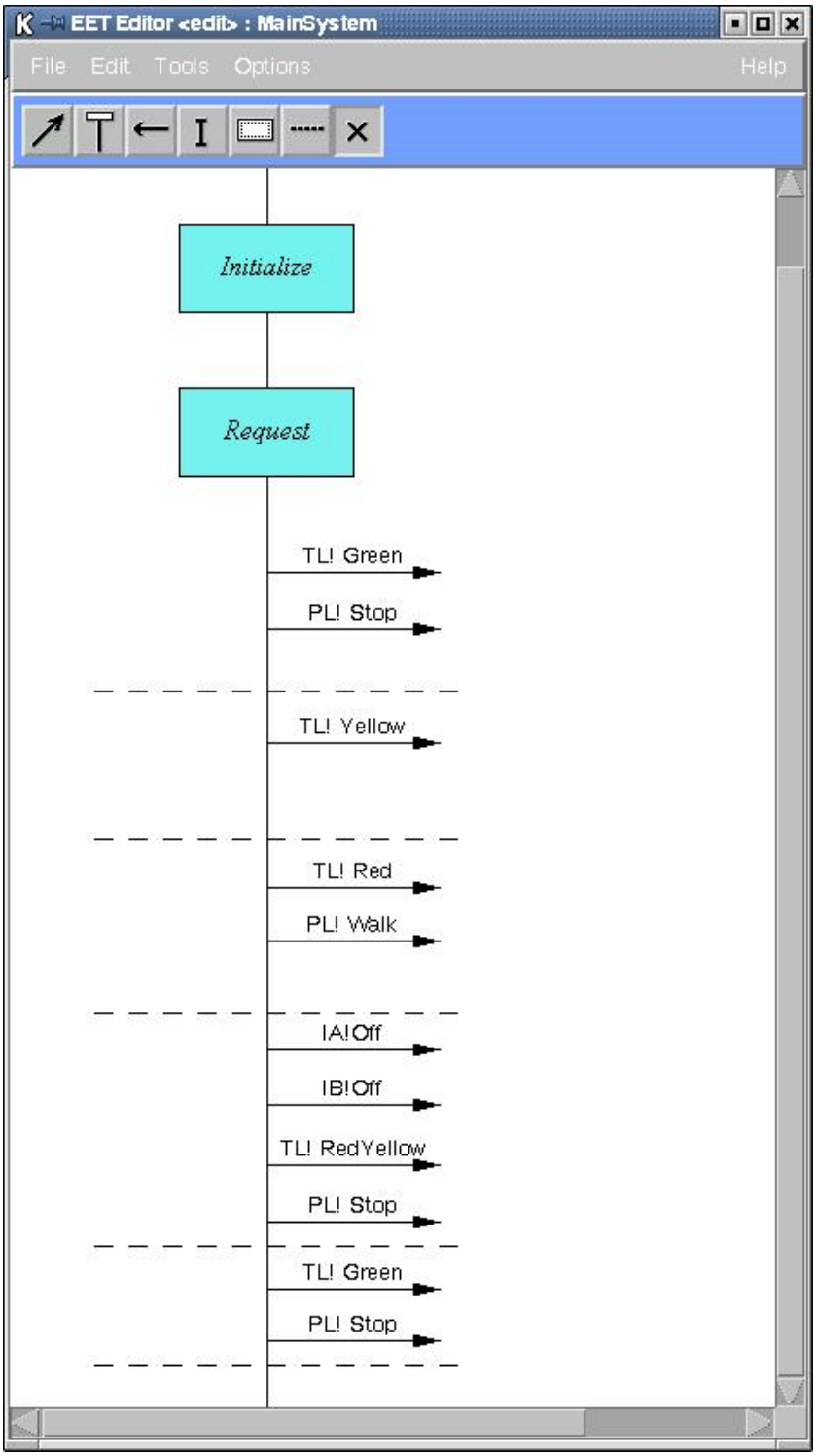

Figure 5.13: The complete EET of the Facility component

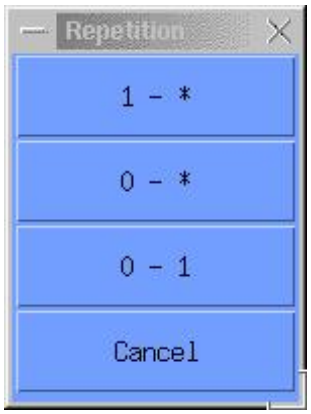

Figure 5.14: Select box for the indicator label

fashion. Here, two different runs are possible: in the first case, the signal comes from button A, in the second case from button B. Insert both runs in the `Request` box.

Now you have completed the interaction diagram for the traffic lights controller. Save all five documents, using ”File -> Save & Close”.

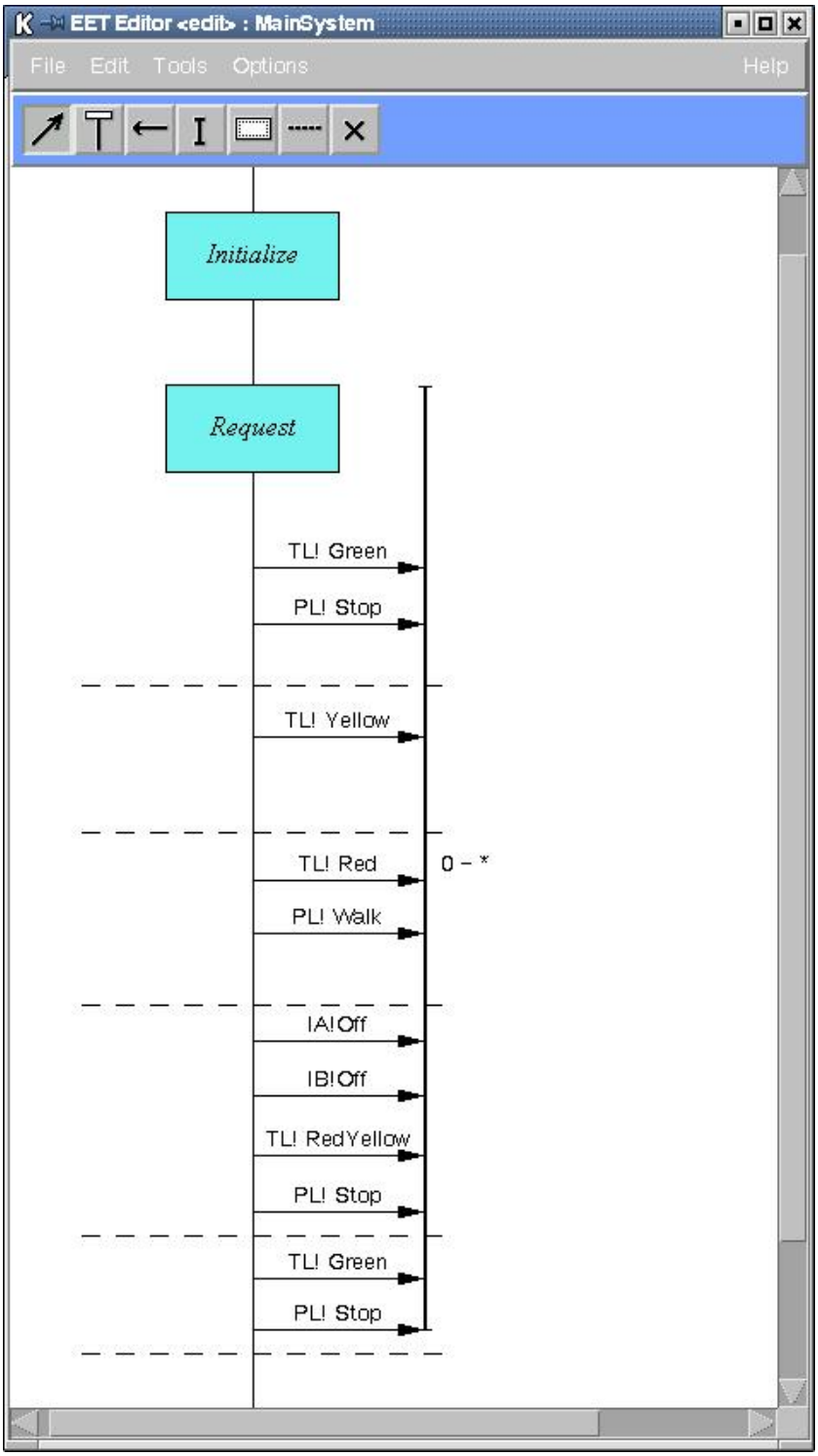
EET Editor <edit> : MainSystem
File
Edit
Tools
Options
Help
Initialize
Request
TL! Green
PL! Stop
TL! Yellow
TL! Red
0 – *
PL! Walk
IA!Off
IB!Off
TL! RedYellow
PL! Stop
TL! Green
PL! Stop

Figure 5.15:

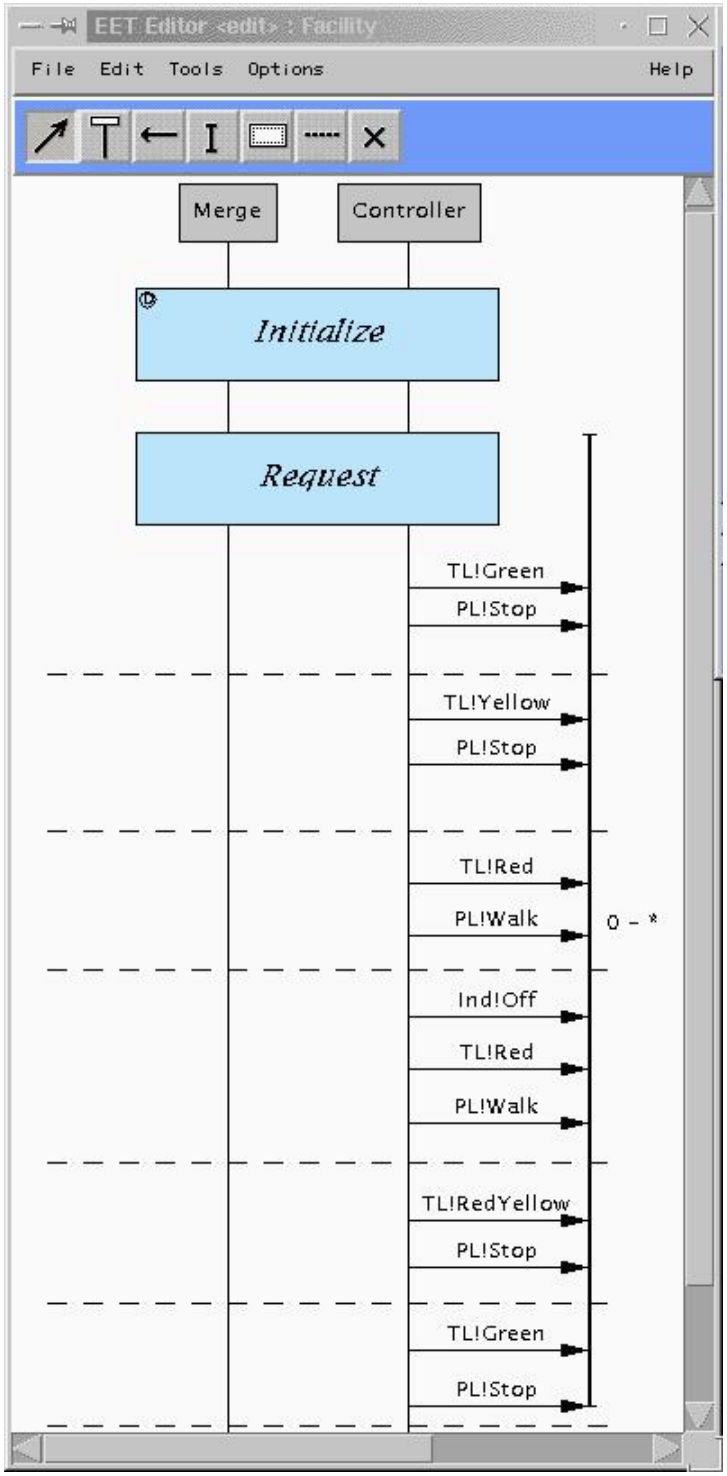

Figure 5.16: Decomposing the interaction of the ”Facility” component into two components, ”Controller” and ”Merge”

Figure 5.17:

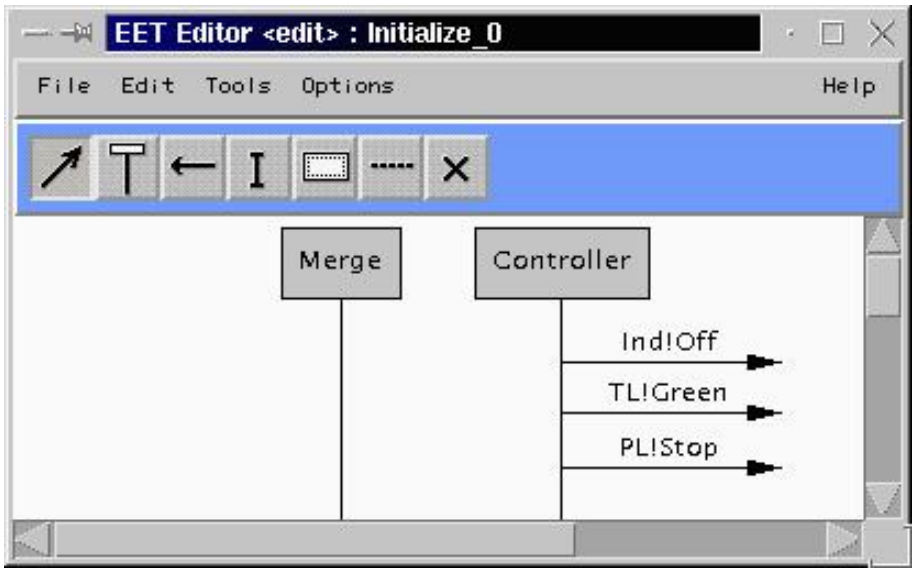

Figure 5.18:

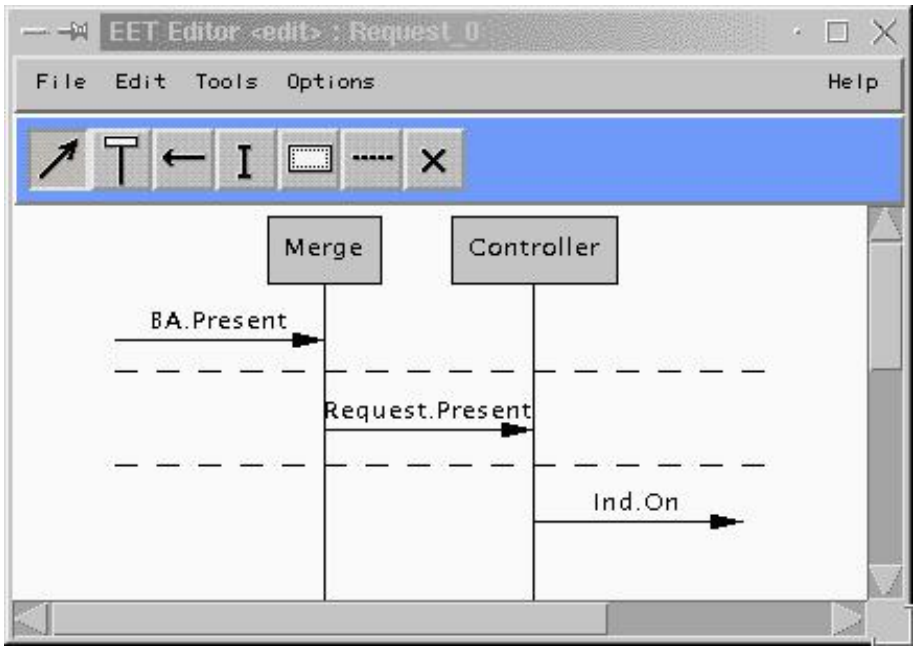

Figure 5.19:

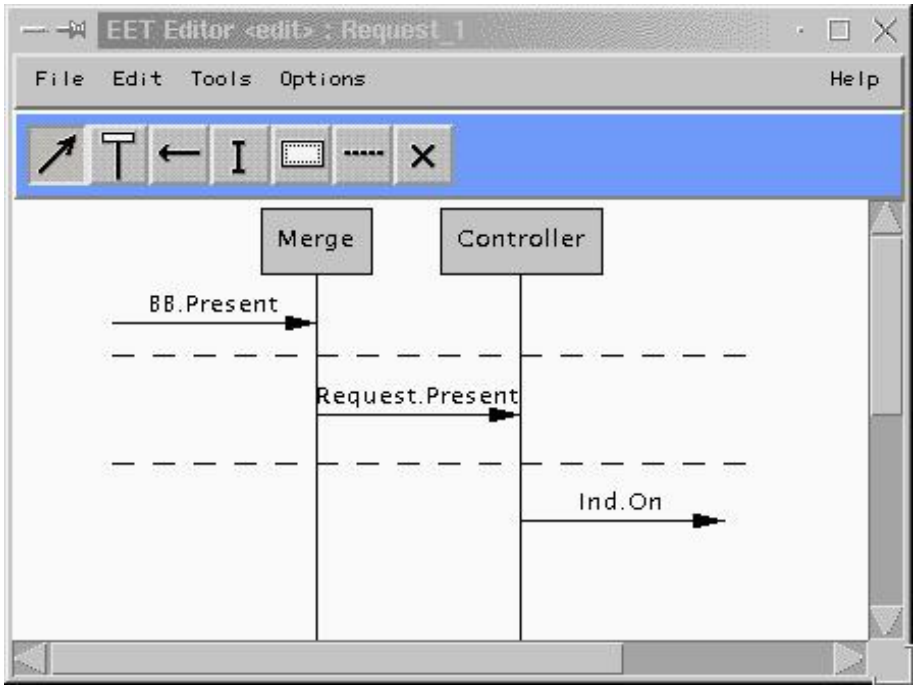

Figure 5.20:

Chapter 6

Behavioral View

Specifying how the entire system is divided into components and how they are interconnected, the structural view defines only the static structure of the system. A preliminary dynamic outline of the system behavior is provided in interaction diagrams. This is done in an exemplary fashion by demonstrating the chronological order of certain signals and messages.

However, this is not yet sufficient to implement the system - it must be specified in complete detail how each component reacts to incoming messages, which new messages are generated, and where they are sent to. As a rule, the behavior of the individual components is specified by means of state transition diagrams. We will explain this in the following.

What do we mean by a state? To illustrate the concept, one might say that a state represents important properties of a component. These may be properties which are visible externally, but also inner properties. The state of a traffic lights controller no doubt comprises the messages on its input ports, as well as the messages on its output ports which it currently transmits in order to influence the environment. However, there is probably an inner clock ”ticking” in such a controller, by means of which the intervals between the traffic light phases are set. If this clock, starting from an initial value, counts down the time steps to 0, the state of the control system changes with each time step, even if this is not necessarily visible outside.

In order to store and transmit data, values are assigned to parameters. A system thus possesses a finite set of parameters. Technically speaking, these are the input and output ports and, in addition, the local variables of a component. The latter will be explained in detail in the following chapter.

Moreover, there is a finite number of situations which are typical of the component’s behavior. The component is always in precisely one of these situations. In

the traffic lights controller example, the four phases red, yellow, green, and red-yellow might be regarded as such situations. Now we can be more precise: by a state we mean a pair consisting of

- a current valuation of the parameters (data state) and
- a typical situation (control state).

What makes this kind of modeling particularly challenging is the question which situations are to be considered typical. We call the set of all possible states of a component its state space S. Note that S is infinite although only a finite number of parameters and situations are used as soon as an infinite number of parameter valuations is permitted.

As mentioned, the state of a component does not remain one and the same. After all, ”behavior” is all about ”reacting to causes” and, therefore, ”changing the state.” Thus we have to determine on what conditions our component changes from one state to another. We want to provide for the possibility that there is a choice of several successor states for a state s with the conditions remaining the same. This can be described by means of a state transition function Z: S -> P(S). It is a mapping to the set P(S) of the subsets of S (the so-called power set of S). The set then contains all valid successor states of s.

Now we can define as follows: given a state set S, a non-deterministic state transition system is given by

- an initial state a from S and
- a state transition function Z.

The fact that these are state transition systems with inputs and outputs is expressed by the valuations of the parameters which represent input or output ports.

Let us now have a look at a sequence of states which begins with the initial state and in which each state, unless it is the last one in the sequence, is followed by a (randomly chosen) legal successor state. The succession may be interpreted as ”later.” This means that we are faced with something like a time behavior, of course without a defined measurement for duration. We talk about a run of the system or also about a causal chain.

So how is the transition function Z specified? Transition diagrams are a convenient means of description. They represent the control states by nodes and the transitions by means of directed edges between the nodes. To obtain a complete definition of Z, each edge included in the diagram must be accompanied by text, indicating which data states the edge is provided for. We will see soon that AutoFOCUS opens a helpful input window for each edge to make such specifications easier. The edges have names.

After all, we are interested in modeling the behavior of the individual system component. Since we intend to use one transition diagram in each case, we have to think about control states which carry the data states ”piggyback” so to speak. With this trick, even components with an infinite number of states (an infinite variety of data) can be tackled by means of finite descriptions (finite diagrams).

6.1 State Transition Diagrams (STDs)

In distributed systems, the behavior of components is described by means of *State Transition Diagrams*. An STD describes the behavior of a system or a component in terms of its reactions to inputs from the environment and the resulting outputs to the environment. The reactions depend on the current state of the component. Visually, STDs are represented by graphs with oval nodes for the states and annotated arrows for the transitions.

6.1.1 Control States

The possible control states of the state transition system are described by the nodes (labeled ellipses) in the STD. The label is merely a name and has no influence on the behavior. In every STD, precisely one state must be marked as the *initial state*. This is the state in which the system is when it starts. The initial state is identified by a black circle on the left-hand side of the ellipse.

Example:
Typical situations in which our set of traffic lights may be result from its displays. Therefore, we determine that the `Controller` component shall always be in one of the following states: `Green`, `Yellow`, `Red`, `RedYellow`. In addition, there is a state for the initialization (`Init`). Figure 6.1 shows the states of the Controller.

6.1.2 Local Variables

Each system component may have (any number of) *local variables*. Local variables have a name and an assigned type. They are called local because each of them is assigned to precisely one component. In AutoFOCUS they are, therefore, declared with the respective component in the system structure (SSD). Only the state transition system associated with the component can access the local variables, i.e. read them or write to them. In the local variables, data can be held

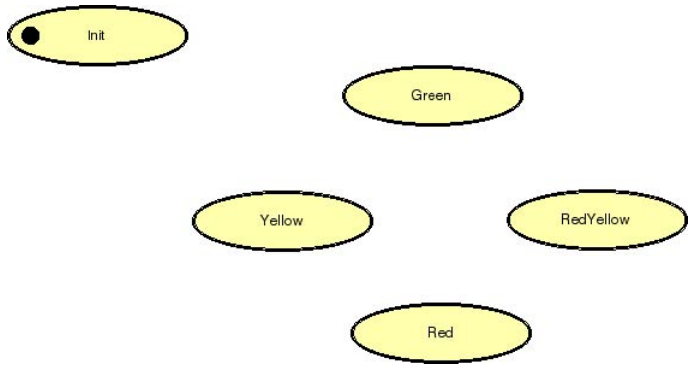

Figure 6.1: Figure 1: States of the Controller component

independent of the control state. Local variables in AutoFOCUS correspond to the internal parameters of a state transition system, as described in the chapter Behavioral View (see Chapter 6). A valuation of all local variables of a component is called data state.

6.1.3 Data states

In addition to the control state in which the state transition system currently is and which is represented by a node of the graph in the STD, there always exists a data state. The data state is represented by the values of the internal parameters (local variables) and external parameters (ports) of the component. The *entire state* in which a component can be always consists of both the control state and the data state.

Example:
In addition to the control state, the traffic lights controller has a local variable `t` which is used as a timer. If the traffic lights switch to a new control state (e.g. Yellow), the variable `t` is to be set to a certain value and then continuously to be reduced by 1. As soon as `t == 0` is reached, the traffic lights switch to the next control state. Thus, even though the `Controller` remains in the `Yellow` control state, the data state, and consequently the entire state, is changing continuously.

6.1.4 Transitions

A transition describes how a state transitions into another state. Represented by directed edges in the STD, transitions always connect two control states: the source

state and the target state. That said, the source state need not necessarily differ from the target state. In addition, a transition can be described more precisely with the following attributes:

- a precondition
- a set of input patterns
- a set of output patterns
- a set of postconditions
- a label
- a comment

A transition can fire (i.e. can be executed) precisely at the moment when the assigned component is in the source state of the transition, the precondition is fulfilled, and the inputs on the input ports can be matched with the input patterns. After the transition has fired, the control state of the component turns into the target state of the transition, the data state is adapted in accordance with the given postconditions, and the output is written to the output ports in accordance with the output patterns. The names assigned to the transitions solely serve as labels in the diagram. The comment may contain further hints regarding this transition.

STDs can be non-deterministic. This means that in principle, different transitions may fire at a given point of time. In a concrete run, one of the possible transitions is selected and fires in the current situation. If the same situation occurs again at a later point of time, it may well happen that another transition is chosen out of those which are possible in principle.

Example:
Figure 6.2 shows the state transition diagram of the Controller. The names given to the transitions are simple labels. The table below demonstrates the annotations of the `Receive request` transition. The thin ellipses represent transitions with identical source and target states. (Arrowheads have no significance here and may, therefore, be left out.) Annotations of the `Receive request`-transition:

Field name	**Value**
precondition	`t == (0 - 1)`
input pattern	`Req?Present`
output pattern	`TL!Green; PL!Stop; Ind!On`
postcondition	`t = TGreen`
label	`Receive request`
comment	

The precise meaning and the implications of the various fields are described in the following.

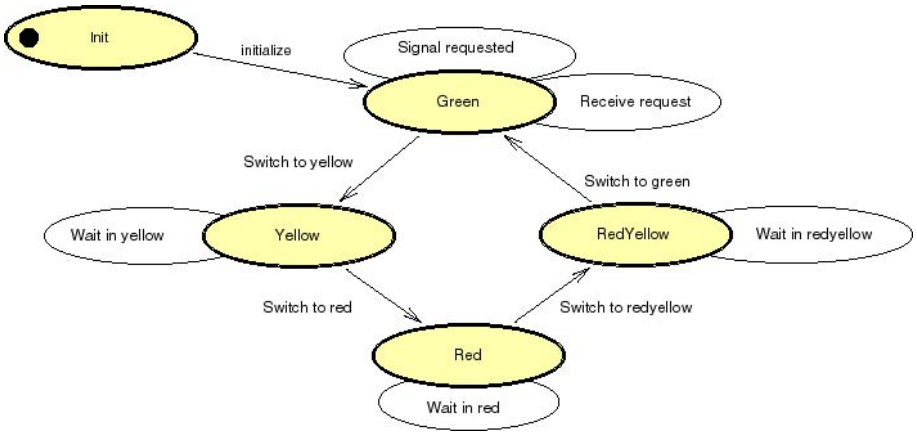

Figure 6.2: Figure 2: State transition diagram of the Controller

6.1.5 Pattern Parameters

Pattern parameters are bound to values by means of matching (see Section 4.4.7) of input patterns. The pattern parameters can then be used with their current values in the precondition, the output patterns, and the postconditions.

6.1.6 Precondition

For a transition to be able to fire, its precondition has to be fulfilled, i.e. its evaluation must yield `True`. The precondition is essentially a boolean term of the programming language used in each case (Java or QuestF). In the precondition, pattern parameters and local variables of the associated component may be used.

Example:
In the transition annotation below, a pattern parameter `m` is defined which is bound to the value that is present on port `input`. This value must be greater than zero for the transition to fire. In the postcondition, the value read previously is stored in the `lokal`local variable of the associated component.

```
precondition:   m>0
input pattern:  input?m
postcondition:  lokal = m
```

6.1.7 Input/Output Patterns

In the input and output patterns, the behavior is defined which is visible from outside. A transition can only fire if the values which are present on the ports match the input patterns. By means of output patterns, new values are determined which are written to the output ports after the transition has fired.

An input pattern consists of the name of an input port and a pattern which may contain local variables of the associated component or pattern parameters. The name of the port and the pattern are separated by a question mark (`?`). If several input patterns are to be checked at the same time, they can be written one behind the other, each separated by a colon (`;`). A message which is present on an input port matches an input pattern if pattern matching between the message and the available pattern succeeds. Before matching, local variables are replaced by constants which correspond to their values, and pattern variables are bound to values corresponding to those present on the port (see also the section on Pattern Comparison (see Section 4.4.7)). Ports not contained in any input pattern of a transition are not checked, i.e. any value can be present on them. If, by contrast, the non-existence of signals is explicitly desired, this may be demanded by providing an empty pattern.

Remark: If Java is used, input patterns must either contain single constants, pattern parameters, or local variables. Here, QuestF offers the additional possibility of checking the input for certain patterns by means of pattern matching and binding one or several sample parameters.

Example:

The input pattern

```
Req?Present
```

of the ”Receive request” transition calls for the `Present` signal on port `Req`.

The input pattern

```
Date?TMJ(1,monat,jahr)
```

calls for a signal of the form `TMJ (1,x,y)` on port `Date`, with x and y being arbitrary integers. If the transition fires, `monat` is bound to the value of `x` and `jahr` to the value of `y`. A port can also be checked for the non-existence of signals. For this, an empty pattern is used. A transition with the input pattern

```
UserInput?
```

can only fire if no signal is present on port `UserInput`.

Output patterns are pairs of names of output ports and expressions which must be of the same type. The name of the output port and the associated expression are separated by an exclamation mark (`!`). Just as is the case with input patterns, several output patterns can be named, separated by a colon (`;`). In addition to all legal operators and functions, the expressions may also contain local variables of the associated component and pattern parameters from the input patterns. If local variables are used, the values which they were bound to before the transition fired are used on principle. No signal is issued on ports which are not contained in the output patterns. In this case, any signal that may have been written earlier is deleted.

Example:

The transition label

```
Eingabemuster:  Req?Present
Ausgabemuster:  TL!Green; PL!Stop; Ind!On
```

of the `Receive request` transition calls for the `Present` signal on port `Req` and sends the signals `Green` to port `TL`, `Stop` to `PL` and `On` to `Ind`.

6.1.8 Postcondition

The data state of the component is modified by means of the postcondition. Even though it is called ”condition”, it is no predicate but an assignment of new values to the local variables. These assignments have the following form:

Assignments must be type correct. The expression may contain pattern parameters as well as local variables. If a local variable does not appear on the left-hand side of an assignment, its value is preserved.

Example:
In the "Receive request" transition, the local variable `t` of the associated component is set to the value of the TGreen constant with the assignment

```
postcondition: t = TGreen
```

6.1.9 Modeling the Behavior of the Traffic Lights

The system structure of the traffic lights controller contains two atomic components which are not decomposed further into subcomponents: `Controller` and `Merge`. To ensure that the model is complete (and, therefore, can be simulated), a behavior must be assigned to both components by means of state transition diagrams (STDs). In the following two sections, we describe the STDs of these components

The Merge Component

For pedestrians who wish to cross the street, two pushbuttons to request the signal are provided, one on each side of the street. For the control system, however, it is irrelevant on which side the request is made. The `Merge` component's task is to react to both pushbuttons, and send the corresponding signal to the `Controller`.

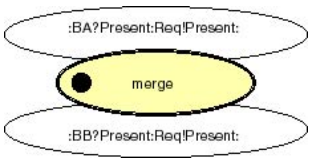

Figure 6.3: Figure 1: Behavioral description of the Merge component

Figure 6.3 shows the corresponding state transition diagram. The upper transition fires if the `Present` signal is present on port `BA`, i.e. if the button on either side of the street has been pushed. As a result, a signal is generated on port `Req` which is sent to the Controller via the `Request` channel. If the other button is pushed, the lower transition can fire, generating a signal on port `Req` as well. If both buttons are pushed at the same time, both transitions could fire in principle. In a concrete run, one of the transitions is selected. However, this doesn't make any difference for the behavior of the `Merge` component, since the externally visible behavior is the same in both cases

The Controller

The actual traffic lights control unit is realized in the `Controller` component. As soon as a signal is read from its `Req` port, the lights for cars shall gradually switch to red and the lights for pedestrians shall turn green. Figure 6.4 shows

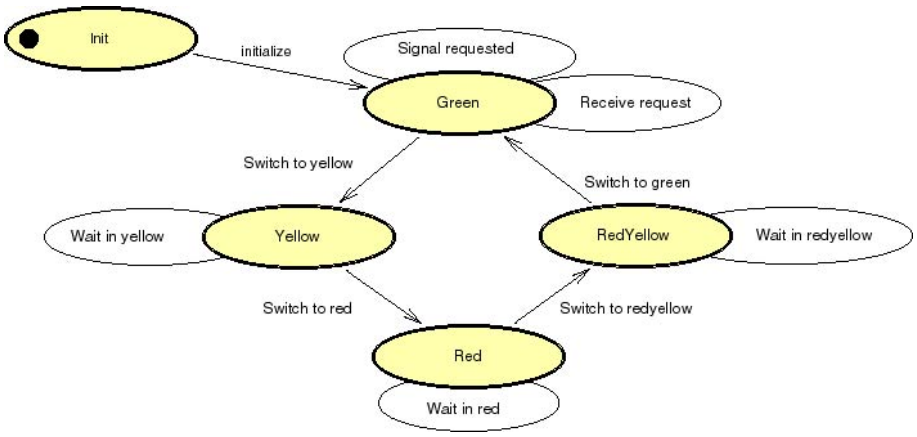

Figure 6.4: Figure 2: Behavioral description of the Controller

the state transition diagram of the `Controller` component, in which a local variable `t` of type `Int` is used. With the aid of this variable, the duration of the individual phases is modeled (e.g. the duration of the green phase for pedestrians). For this purpose, t is set to one of the predefined constants (e.g. `TGreen`) and then gradually counted down by 1, until the value `0` is reached. Then the time limit is expired and the system can switch to the next state. Since `t` is used exclusively in this context, it is also called *timer variable*. A negative value symbolizes that the timer is switched off.. The table below lists the various attributes of the transitions used in Figure 6.4. The column headed ”Label” corresponds to the transition labels in Figure 6.4.

Label	Precondition	Input	Output	Postcondition
initialize			TL!Green; PL!Stop; Ind!Off	t = 0-1
Receive request	t == (0-1)	Req? Present	TL!Green; PL!Stop; Ind!On	t = TGreen
Signal requested	t > 0			t = t - 1
Switch to yellow	t == 0		TL!Yellow	t = TYellow
Wait in yellow	t > 0			t = t - 1
Switch to red	t == 0		TL!Red; PL!Walk; Ind!Off	t = TRed
Wait in red	t > 0			t = t - 1
Switch to redyellow	t == 0		TL!RedYellow; PL!Stop	t = TYellow
Wait in redyellow	t > 0			t = t - 1
Switch to green	t == 0		TL!Green	t = 0 - 1

The `Controller` starts in the `Init` initial state (initial states are marked with a black circle on the left). The `Initialize` transition neither has a precondition, nor is a certain input required via an input pattern. Therefore, this transition fires right after the simulation has started. The task of the `Initialize` transition is to switch the lights for cars to green and the lights for pedestrians to red, to switch off the indicator lights, and to set the local variable `t` (the timer) to `-1` (timer switched off). The traffic light displays are realized by corresponding signals on ports `TL`, `PL` and `Ind`. The timer variable `t`is set to `-1` in the postcondition. After the transition has fired, the component is in the `Green` control state. As soon as a `Present` signal is received by Port `Req`, the `Receive request` transition can fire, setting the timer variable `t` to the value of the `TGreen` constant. By means of the `Signal requested` transition, `t` is counted down by `1` each, until `t == 0`. Then the transition to the `Yellow` control state takes place by means of the `Switch to yellow` transition. At the same time, the lights for cars switch to yellow and the timer variable `t` is set to the value of `TYellow`. In the `Yellow`, `Red` and `RedYellow` control states, the same scenario is going on

all the time: the timer variable `t` is counted down to zero. Then the traffic lights switch to the next control state and reset the timer. Only during the transition from the `RedYellow` control state back to the `Green`control state is the timer not reset; instead, the postcondition `t = 0 - 1` binds the local variable to the value `-1` which means that the timer is switched off. Once the `Controller` is again in the `Green` control state, the cycle described above can repeat itself. Hence, a local variable `t` is required to describe the behavior of the `Controller` component. In AutoFOCUS, local variables are parts of the components. Therefore, their existence and association must be described in the respective SSD. For the `Controller` component, we will catch up on this in the instructions for STDs.

6.2 Instructions for STDs

In the following, we will explain step by step how to create state transition diagrams (STDs) in AutoFOCUS. The aim is to create a complete behavioral view of the traffic lights controller (see Section **??**). For more details about STDs, please see:

- Behavioral View (see Chapter 6)
- State Transition Diagrams (see Section 6.1)

6.2.1 New ”Merge” STD Document

Start AutoFOCUS. The window of the project browser with the *Pedestrian Traffic Lights* project is displayed (the creation of which was explained in the instructions for SSDs (see Section 3.3), QuestF (DTDs) (see Section 4.5), and EETs (see Section 5.3).)

Expand the list of the various views of the *Pedestrian Traffic Lights* project and select ”STD.” In AutoFOCUS, state transition diagrams are defined and filed in STD documents. To create a new STD document, select ”New” from the ”Document” menu. A dialog window named ”New STD” is displayed, and you are asked to enter the name of the new document. Type *Merge*. Now the STD editor opens.

6.2.2 STD Editor

The window of the STD editor consists of a menu bar, a tool bar, and the drawing space.

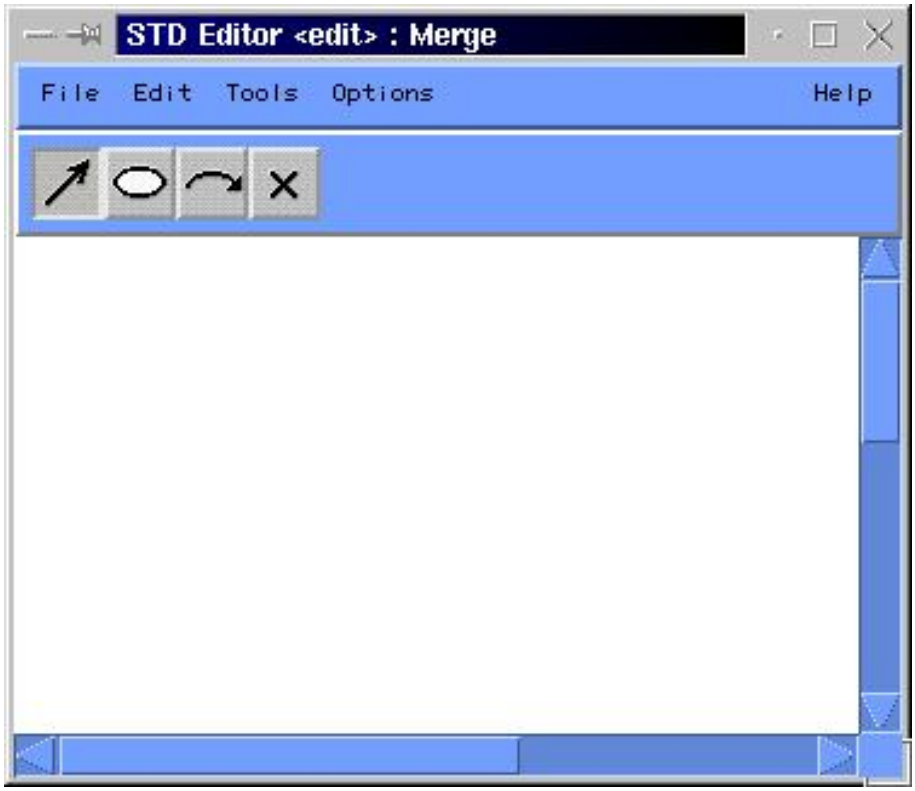

Figure 6.5: Figure 1: The STD editor with the new ”Merge” document

The tool bar symbols have the following functions (the respective function is selected by clicking; precisely one function is active at any given point of time):

Symbol	Name	Function
	Select	selects individual elements of the drawing (highlighting them red)
	State	creates new states
	Transition	creates new transitions
	Delete	deletes elements from the diagram by clicking them with the mouse

For general advice on how to use the editors in AutoFOCUS, consult the operating instructions for editors. (see Section 7).

First we describe the behavior of the `Merge` component in a simple state transition diagram. Next, we create a more complex STD to describe the behavior of the `Controller` component.

6.2.3 Inserting a Control State

The behavior of the `Merge` component can be described by means of a single control state and two transitions. First, we insert a new control state: select the ”State” function from the tool bar and click into the drawing space, at the spot where the new control state is to be located. A ”Set Name” dialog window opens.

Call the state *merge*. Since it is also the initial state, set the kind of the state to ”initial” and confirm by clicking ”Ok.” The new control state is now displayed in the STD editor.

6.2.4 Inserting a Transition

To create transitions in AutoFOCUS, select the ”Transition” function from the tool bar. Then click inside the source state (in our case the `merge` control state) and drag the mouse cursor into the target state (here, back into the `merge` state, drawing a loop). After that, a dialog window named ”Set Label” is displayed, in which the *annotations* of the transition are entered. Here the precondition, input

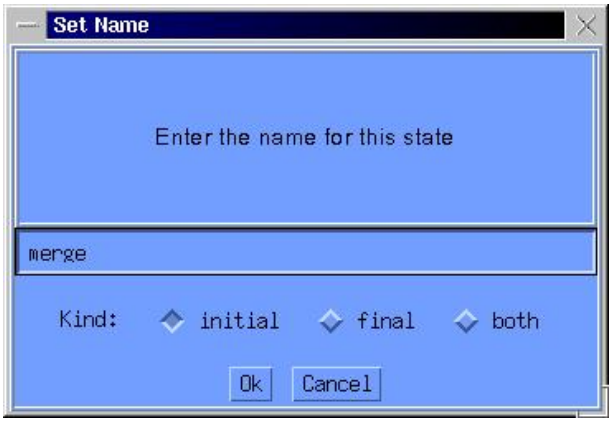

Figure 6.6: Figure 2: Dialog window for entering the document name

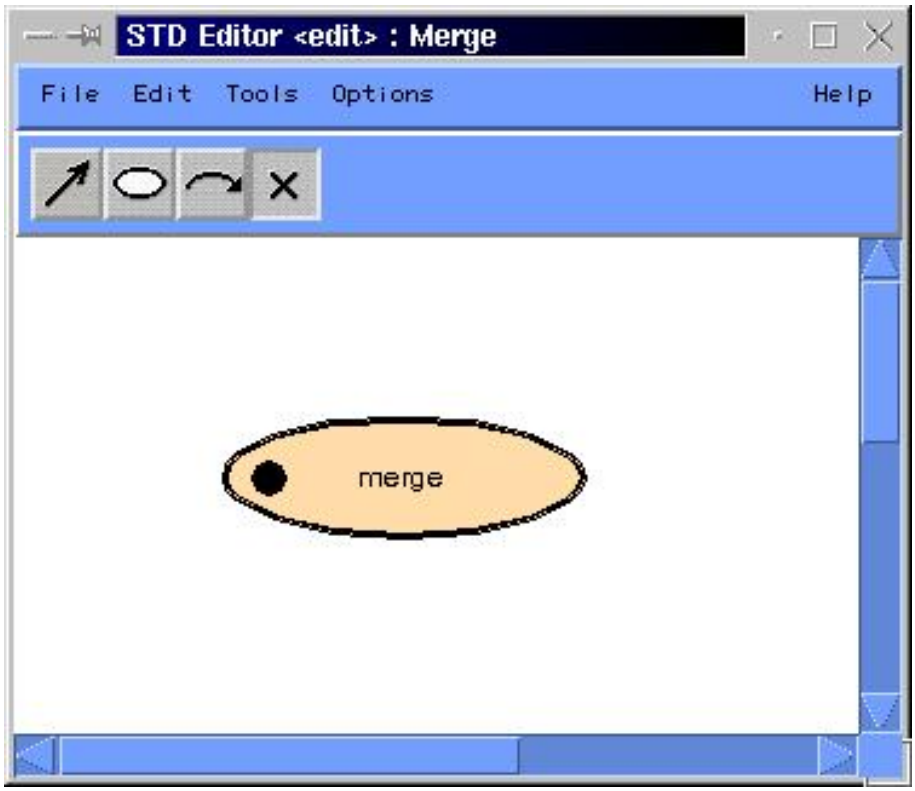

Figure 6.7: Figure 3: The STD editor with one state

pattern, output pattern, and postcondition can be named if required. Moreover, it is possible to add an alternate name as a label for the transition and a comment. The label is displayed as the transition's marker in the STD editor. If the cursor is placed above a transition in the STD editor, the precondition, input, output, and postcondition appear in balloon, separated by colons. If no separate label is given, the balloon text is displayed as the label in the STD editor right away.

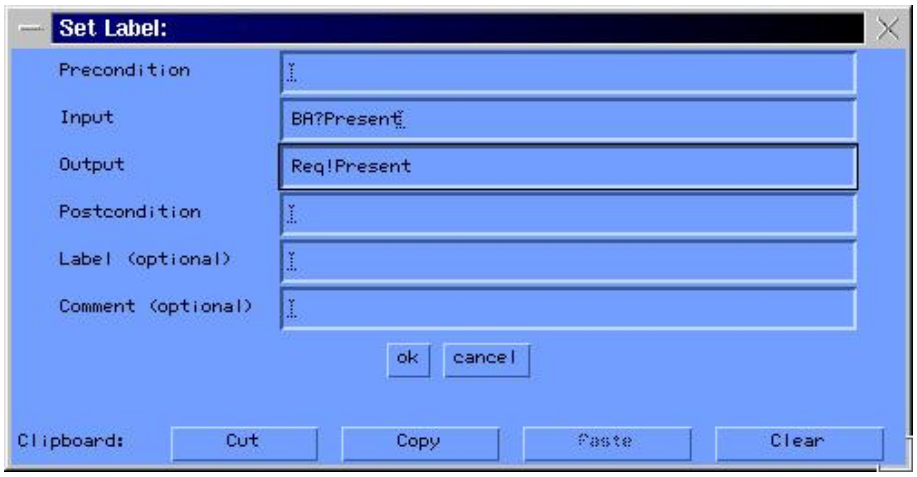

Figure 6.8: Figure 4: Dialog window for setting the label

Fill in the dialog window as shown in Figure 6.8. This procedure determines that the transition fires without a precondition as soon as the `Present` message arrives at port `BA`, binding channel `Req` to the `Present` message. No assignments are made in the postcondition. Before closing the dialog window, we make use of the *clipboard*. Since the label of our second transition differs from the first one only by one single character, we copy the content into the clipboard by clicking "Copy" and close the dialog window by clicking "Ok."

The second transition is attached to the `merge` control state as well. It differs from the first one only in that the input is expected from port `BB` instead of port `BA`. Again, we draw a loop with the mouse cursor (with the transition function activated) which begins and ends in the `merge` control state. In the "Set Label" dialog window, we click "Paste" to insert the clipboard content. Change the port name in the input line to `BB` and close the window again.

Now your state transition diagram should look like in the figure above. The behavior of the "Merge" component is now fully described. Save the STD document with the "Save & Close" command from the "File" menu.

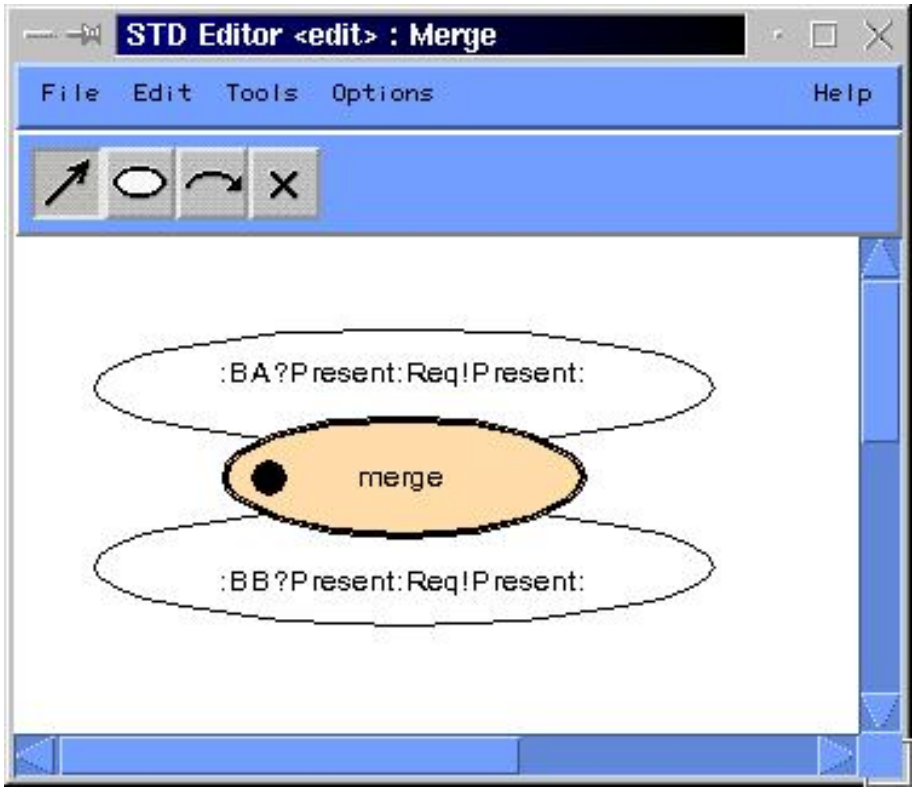

Figure 6.9: Figure 5 The state transition diagram of the ”Merge” component with both transitions

6.2.5 Controller

Describing the behavior of the `Controller` component is a somewhat more ambitious task. We use an initial state ”Init” and one control state each for the four signals of the traffic lights (`none`, i.e. neither initial nor final state). First, create a new STD document named *Controller* in the project browser, proceeding in the same manner as before. In this document, create five control states as shown in the figure below:

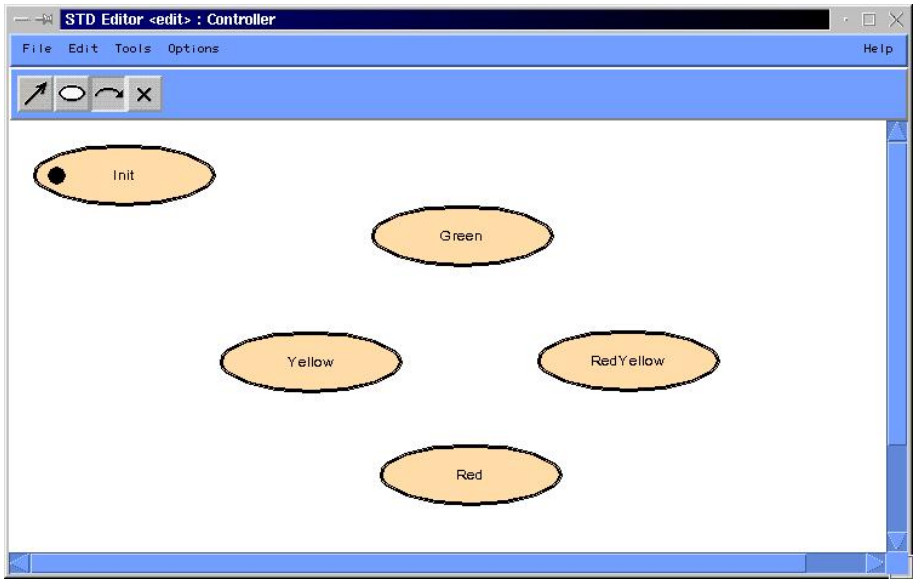

Figure 6.10: Figure 6: The five states of the ”Controller” state transition diagram

Next, we have to create the transitions. We start with the transition from the `Init` control state to `Green`. Use the mouse to drag the cursor from within `Init` to the interior of `Green` with the transition function activated and fill in the ”Set Label” dialog as shown in the figure below:

What is interesting about this transition is its postcondition. When the transition is executed, a local variable `t` (the timer variable which will be defined later on) is set to the value `-1`. The values entered are adopted by clicking ”Ok.”

Now add the other transitions. The required data is listed in the table below:

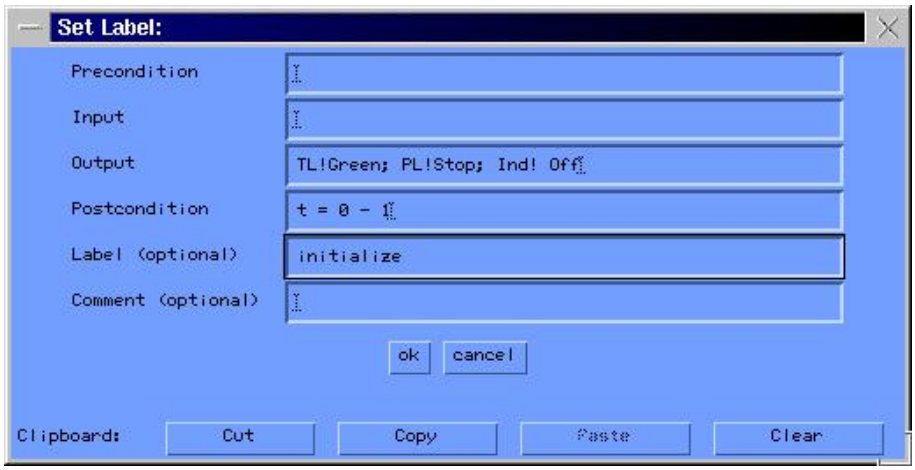

Figure 6.11: Figure 7: Dialog window for setting the Init label

Source State -> Target State	**Precond.**	**Input, Output**	**Postcond.**	**Label**
Green -> Green	t == (0-1)	Req?Present; TL!Green; PL!Stop; Ind!On	t = TGreen	Receive request
Green -> Green	t > 0		t = t - 1	Signal requested
Green -> Yellow	t == 0	TL!Yellow	t = TYellow	Switch to yellow
Yellow -> Yellow	t > 0		t = t - 1	Wait in yellow
Yellow -> Red	t == 0	TL!Red; PL!Walk; Ind!Off	t = TRed	Switch to red
Red -> Red	t > 0		t = t - 1	Wait in red
Red -> RedYellow	t == 0	TL!RedYellow; PL!Stop	t = TYellow	Switch to redyellow
RedYellow -> RedYellow	t > 0		t = t - 1	Wait in redyellow
RedYellow -> Green	t == 0	TL!Green	t = 0 - 1	Switch to green

Now your ”Controller” STD should look roughly like this:

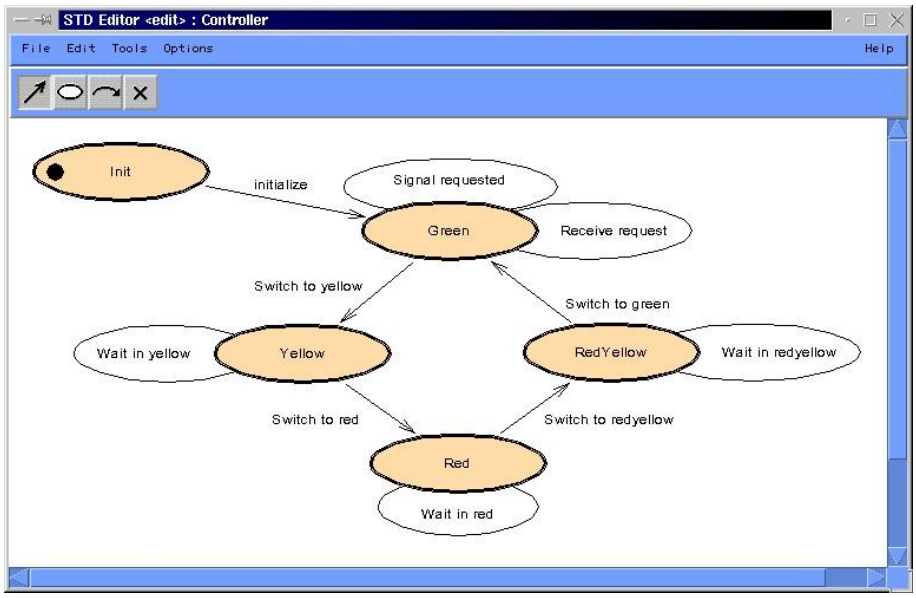

Figure 6.12: Figure 8: The state transition diagram of the Controller

6.2.6 Local Variable

For an executable simulation, the `Controller` component also needs a local variable `t` for the timer, as was explained in the ”Controller (see 6.2.5)” section. Local variables are defined in the associated system structure diagram and are available in all STDs which describe the behavior of the respective component.

Proceed as follows:

1. Expand the ”SSD” entry below the *Pedestrian Traffic Lights* project in the project browser. The ”Mainsystem” component is displayed.
2. Expand the ”Mainsystem” entry. The refined ”Facility” SSD of ”Mainsystem” with the ”Merge” and ”Controller” subcomponents is displayed.
3. After double-clicking ”Facility” in the SSD editor, select the ”Controller” component.
4. Select the ”Local Variables” command from the ”Edit” menu.
5. In the ”CDD Editor” (component data definition) dialog window, set ”Name” to `t`, ”Type” to `Int` (for integer) and ”Init value” to `0-1`. Add the variable to the `Controller` Controller component by clicking ”Add” and leave the window by clicking ”Ok.”

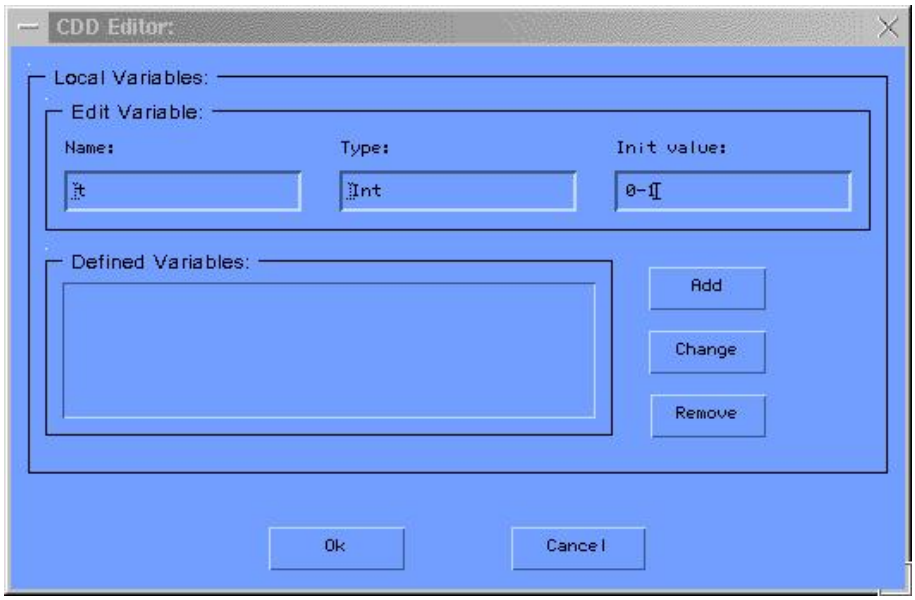

Figure 6.13: Figure 9: ”CDD Editor” dialog window for defining local variables

Leave the SSD editor of the ”Controller” component open to carry out the next step.

6.2.7 Linking SSDs with STDs

Finally, you have to link the system structure diagrams of the `Merge` and `Controller` components with their STDs which you just created. In this way, the behavioral description is assigned to the STDs.

We start with the `Controller` component because it is probably still selected in the `Facility` SSD. Proceed as follows:

1. If necessary, select the `Controller` component in the SSD editor again.
2. Execute the ”Assign STD” command from the ”Edit -> Associations” menu.
3. A dialog window with the available STDs of the model opens. Select ”Controller” and confirm by clicking ”OK.”

In the same fashion, assign the ”Merge” STD to the ”Merge” SSD component.

Now you have created a complete model of the traffic lights controller. Next, AutoFOCUS offers the opportunity to carry out consistency checks (see Section 8.1) and to simulate (see Section 8.2) the model.

Chapter 7

How to Use the Graphical Editors

7.1 Selecting

It is possible to select either one or several elements in AutoFOCUS. There are several ways to do this:

Command Elements	Action
left mouse key	selects an individual element
shift + left mouse key	adds an element to or removes it from the selection
drag with the left mouse key (lasso)	selects all elements in the lassoed rectangular section
shift + lasso with left mouse key	adds an element in the section to the selection or removes it

7.2 Shifting and Changing Size

To shift the depicted elements and to enlarge or shrink them, the following operating functions are available:

Command Elements	Action
press left mouse key on selected element and drag	shifts all selected elements without shifting any elements which have not been selected
middle mouse key on arbitrary element (if available)	shifts an individual element without changing the selection
press left or middle mouse key on the edge of an element and drag	enlarges or shrinks the element as far as this is possible

7.3 Font

The font, font size, and certain attributes such as bold and italic can be changed for all selected elements via the `Edit->Font` menu.

7.4 Clipboard

AutoFOCUS also has a clipboard functionality. Its operation is analogous to the Windows standard:

Command Elements	**Action**
Ctrl-C or Strg-C	copies selected elements into the clipboard
Ctrl-X or Strg-X	moves selected elements to the clipboard
Ctrl-V or Strg-V	inserts the clipboard content into the document

Chapter 8

System Validation

8.1 Consistency Assurance

If a model exceeds the size of a toy example, the information describing the system as a whole is spread over several views and layers of abstraction. This is a major source of errors and inconsistencies.

Since AutoFOCUS uses different as well as hierarchically organized views, it is important to ensure that the information spread over the various views is well-defined or ”consistent” in a methodological sense. Therefore, consistency checks are offered to guarantee that the edited views match. In large projects, it may already be useful to perform consistency checks when certain intermediate stages have been reached.

8.1.1 The Consistency Notion in AutoFOCUS

In AutoFOCUS, several different classes of correctness conditions are subsumed under the notion of consistency:

- Grammatical correctness: the syntactic rules for the textual representation and the graph- grammatical rules for the graphical representation are observed in the various views.
- Interface correctness: if one view is hierarchically embedded into another view, the views must have compatible interfaces (e.g. the input and output ports of the components and their decompositions must be compatible with regard to number, direction, and type).

- Definedness: if elements from one view are used in another view, they must be defined in the respective view (e.g. channel types in SSDs and STDs).
- Type correctness: the type required for using an element must correspond to the element type (e.g. port and value in STD port patterns)
- Completeness: all ”essential” views of a project are available (e.g. each component needs a sub-SSD or an STD for simulation to be possible).

8.1.2 Structure of a Consistency Condition

Each consistency condition consists of the following three parts:

- Name: the name of the consistency condition which is displayed when the conditions to be checked are selected.
- Description: an informal description explaining the purpose of the consistency check. It is displayed in particular if inconsistencies have been found which violate the respective condition. Therefore, the informal description also contains advice regarding the cause and correction of the error.
- Definition: the definition itself is a kind of predicate-logical expression in which objects such as identifiers, components, ports, directions, as well as corresponding functions such as `name_of` may be used.

Remark: Examples of consistency checks can be viewed in the tool. The creation of user-defined consistency checks will not be dealt with in more detail in this tutorial

8.1.3 Integrated Consistency Checks

Several consistency conditions are already integrated in AutoFOCUS. They can be displayed via `Tools->Consistency->Change Check` in the project browser.

8.1.4 Identifying and Correcting Inconsistencies

The consistency check is started in the project browser. Since it is rarely the case that all consistency conditions are already fulfilled during the development of a project, the user may choose an appropriate subset from the list of available conditions. The tool then carries out the selected checks and afterwards shows the list of all inconsistencies found, together with the affected views.

Starting the Consistency Check in AutoFOCUS

Before individual consistency checks can be carried out, the documents must be selected to which the respective consistency check is to be applied. By clicking an entry in the project browser, the complete subtree of this entry (i.e. all documents ranking below the selected document in the hierarchy) is automatically selected for the consistency check. If a complete project is to be checked, the respective project name has to be selected. The consistency check is started via

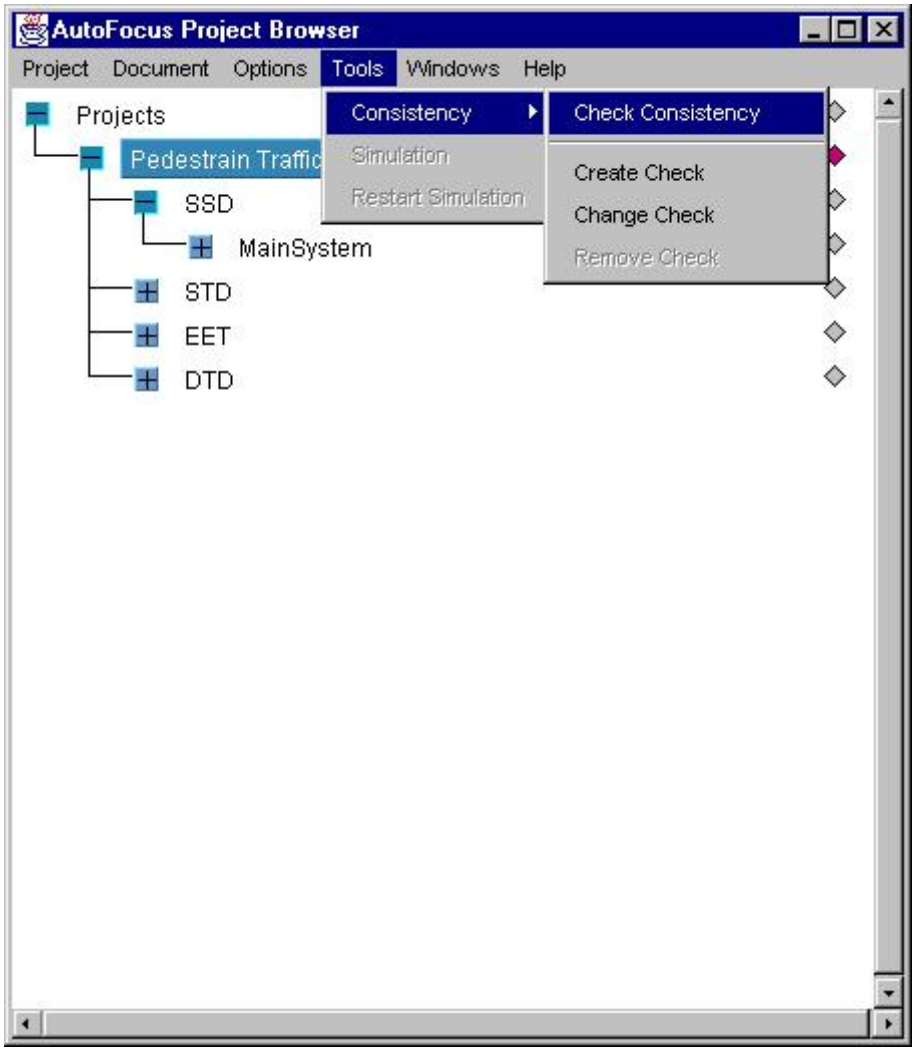

Figure 8.1: Figure 1: Starting the consistency check by selecting the section to be tested

the `Tools->Consistency->Check Consistency` menu item. After the

start, a new window is displayed in which the checks to be performed can be selected. When the consistency checks to be performed have been selected, they are

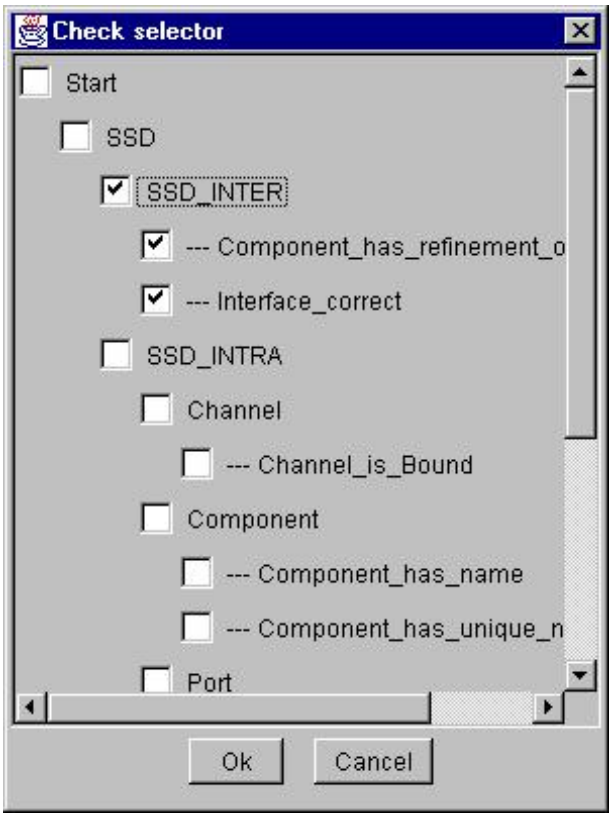

Figure 8.2: Figure 2: The consistency check selector

started. The current status is displayed in another window (see Figure 8.3). When the respective consistency check is finished, the result is displayed in the right-hand column. `yes` means that no inconsistency was discovered (no error), and `no` that an inconsistency exists. The meaning of the individual entries can be looked up in the tool (by means of `Tools->Consistency->Change Check`).

Identified inconsistencies can be displayed graphically. To view the graphics, the failed consistency check has first to be marked. By clicking on Details, a detailed description of the error is presented together with the inconsistencies that have occurred (Figure 8.4). Now the individual inconsistencies can be selected and displayed in the documents by clicking `Details`. The elements which caused the inconsistency are highlighted red. In our example, a port of the `Facility` component had different names in the refined view (`BA` in Figure 8.6) and the abstract view (`ButtonA` in Figure 8.5).

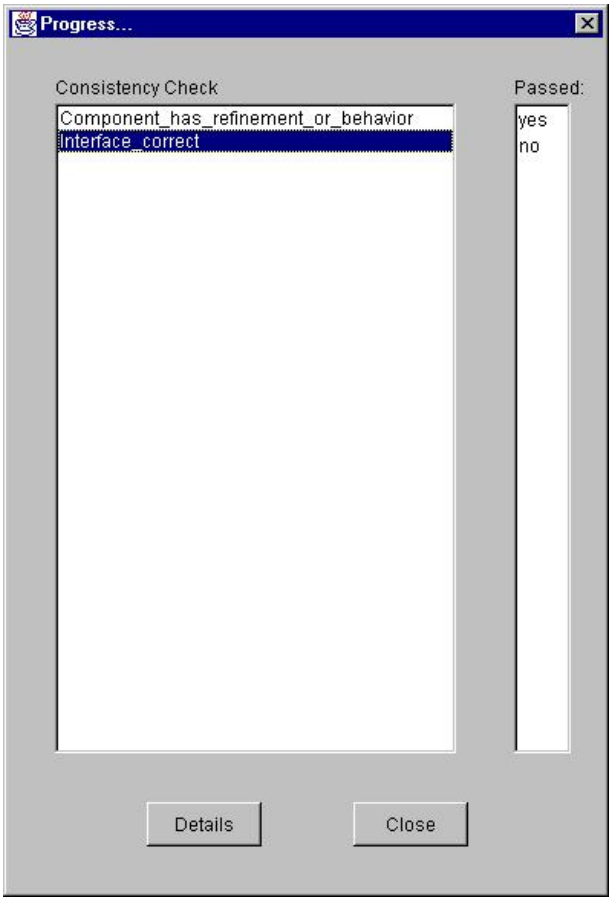

Figure 8.3: Figure 3: Result of the check

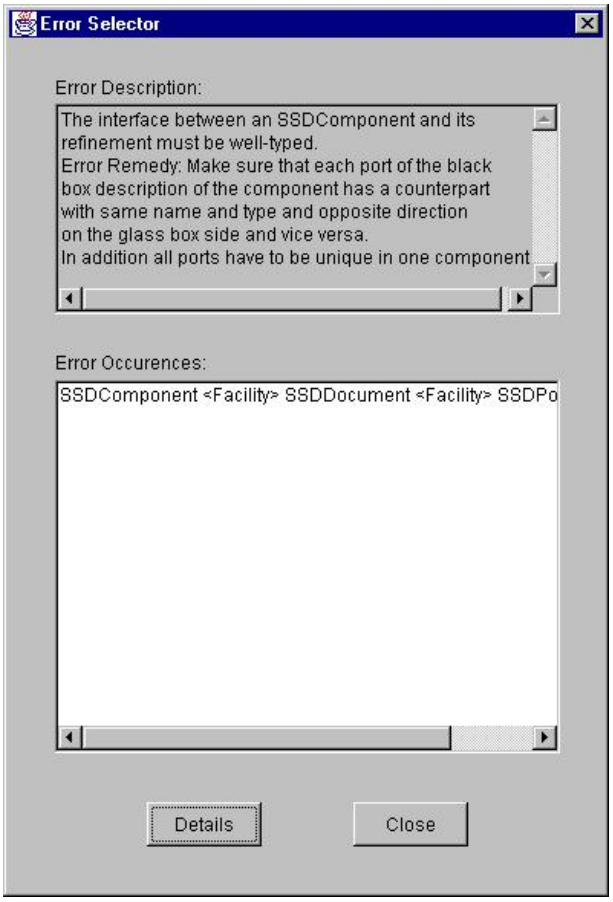

Figure 8.4: Figure 4: Result of the check

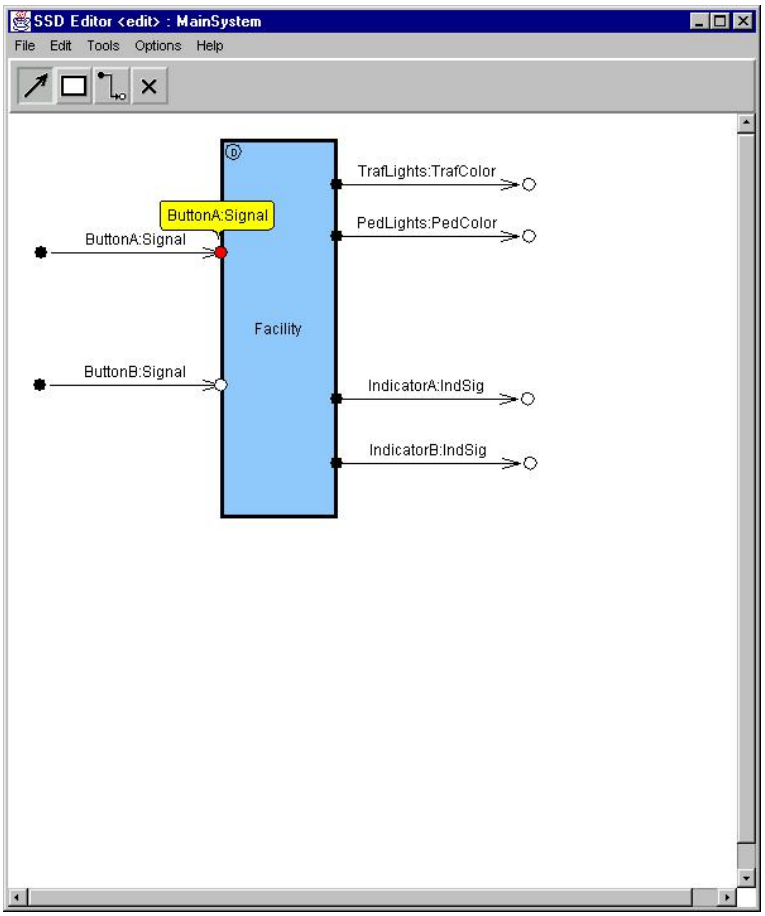

Figure 8.5: Figure 5: Result of the check

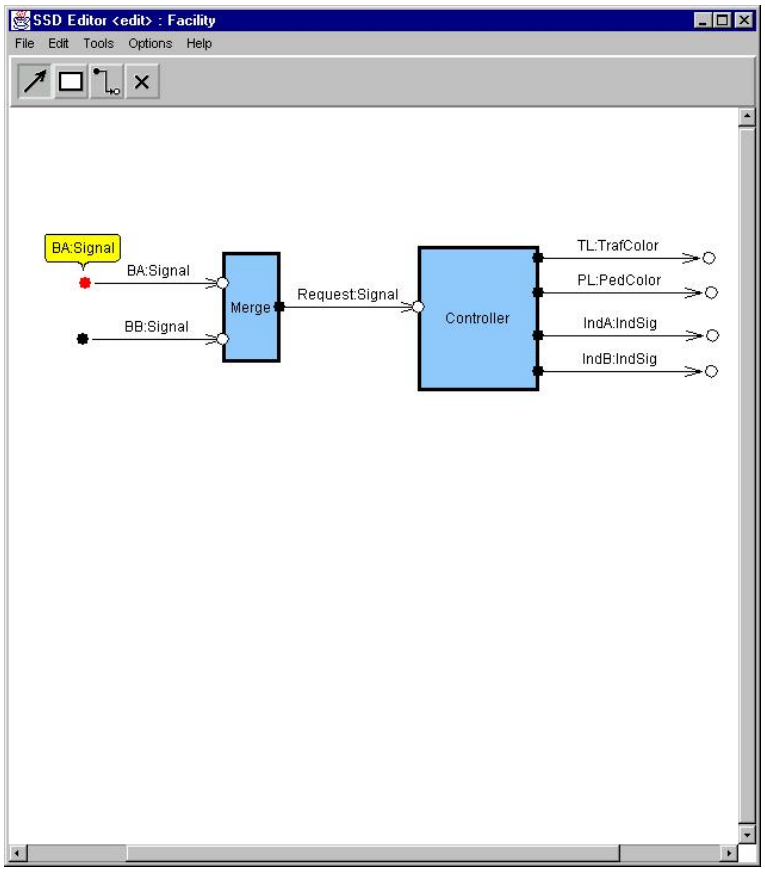

Figure 8.6: Figure 6: Result of the check

Remark: In AutoFOCUS, the type correctness of terms can only be verified in DTDs. If transitions are not type correct, however, this may cause errors during simulation as well.

8.2 Simulation

8.2.1 Overview

Projects which have been modeled in complete detail can be executed in using . It is possible to simulate the behavior of any component whose behavior was previously laid down in a state transition diagram (STD) or which was split up into subcomponents. The simulation consists of animations of the SSDs, the state transition diagrams (STDs), and the EET displays which correspond to the respective run of the system.

If the description of the system behavior is not straight forward at some points, the decision on how to proceed may either be made at random by the system or explicitly by the user. In addition, the user can explicitly enter manual signals which are sent from the environment to the system through input channels. Moreover, the user may decide whether the transitions are to fire with a certain frequency or whether each transition is to be triggered manually.

8.2.2 Checking Consistency

As explained in the previous chapter, AutoFOCUS offers a number of predefined consistency checks. In addition, the user may add consistency checks of his own. It is absolutely essential to perform a consistency check before the first simulation.

To perform the predefined consistency checks, select the name of the project in the project browser and execute the ”Check Consistency” command from the ”Tools -> Consistency” menu. Select the ”Start” button to activate all predefined consistency checks.

8.2.3 Preparation

To simulate our *Pedestrian Traffic Lights* example, start . Select the *Mainsystem* SSD (which contains the `Facility` component) in the project browser.

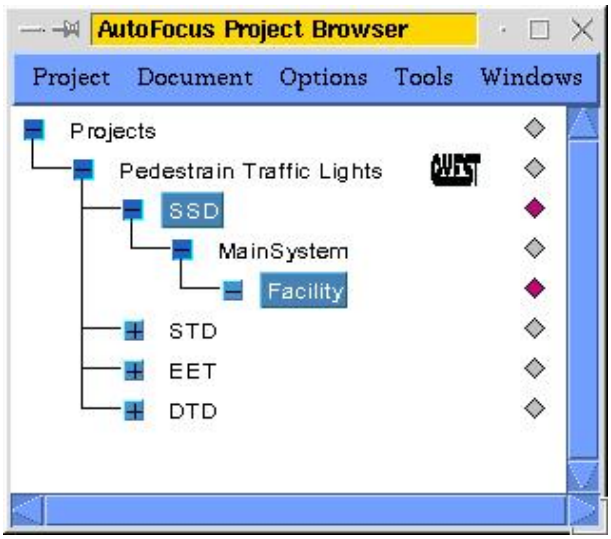

Figure 8.7: Selection of the component to be simulated

Now select ”Simulation” from the ”Tools” menu. Two windows are displayed: The smaller one, ”Starting Simulation”, requests us to select a (sub)system. The systems and subsystems to choose from are shown in the larger window. In our case, we have no real choice because the two subcomponents, ”Controller” and ”Merge”, are simulated together, being the only subsystem of ”Mainsystem” and ”Facility.”

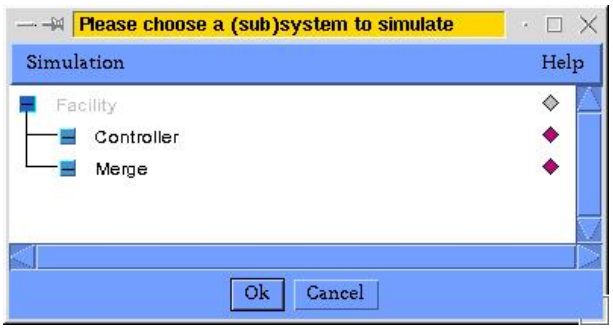

Figure 8.8: Choosing the (sub)system to be simulated

A dialog window, ”Simulation Settings”, is displayed. Here we can decide whether itself shall make a (”random”) choice if several transitions are possible, or whether the user shall make the choice. Select ”ask user”. now generates an executable Java program. Finally, we can decide whether the simulation run is

to be protocolled in an EET. Click ”Yes.”

8.2.4 SimCenter

Now the control window of the simulation environment is displayed.

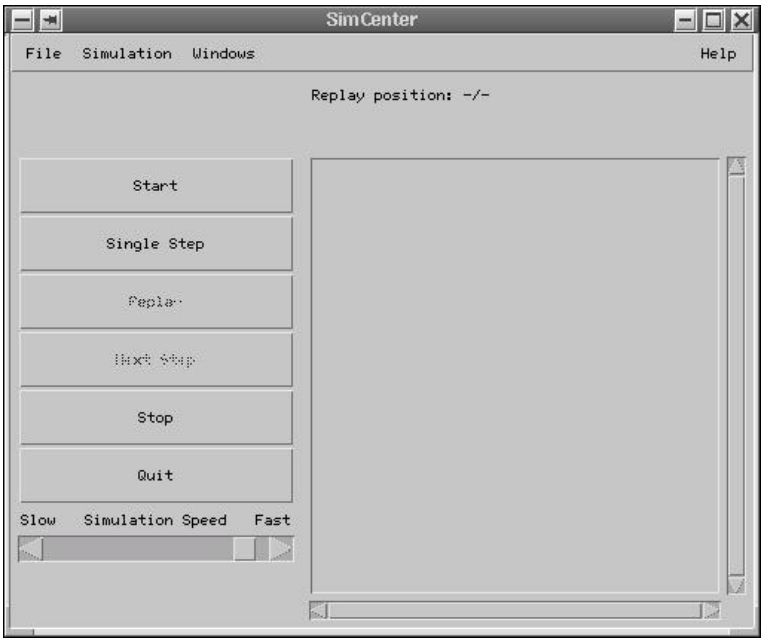

Figure 8.9: The SimCenter simulation environment

First, we select the views we wish to observe during the simulation: in the ”Windows” menu, all system structure diagrams (SSDs), state transition diagrams (STDs), and extended event traces (EETs) which are available in our system can be selected for animation. Open the STD of the Controller and the Facility EET.

In addition, select ”Environment” from the ”Windows” menu of . A table-like window is displayed which shows all channels, ports, and the values on the ports. The user can bind input ports to values during the simulation.

Position the windows on the screen in such a way that all open views are clearly visible.

8.2.5 Simulation

Now start the simulation by clicking on ”Start” in the window. The various transitions fire with a certain frequency. This frequency can be varied using the ”Sim-

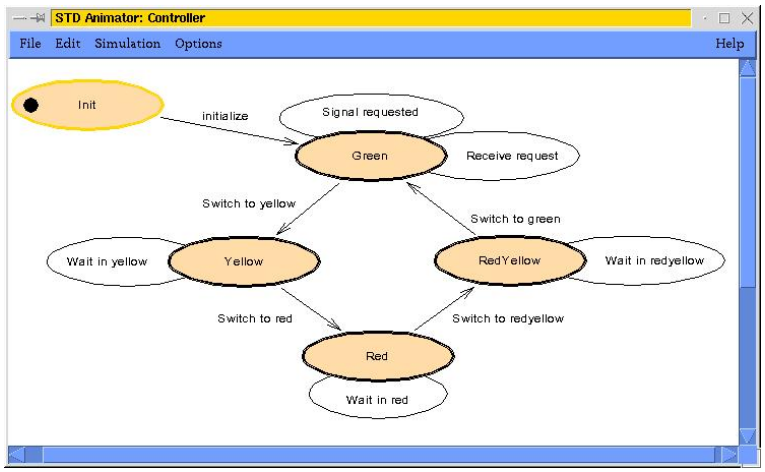

Figure 8.10: The STD of the Controller. The automat is in the ”Init” initial state.

Figure 8.11: The environment window

ulation Speed" scroll bar.

In the SSD of the Controller, the component's current control state and the transition currently firing are highlighted yellow. After leaving the "Init" initial state, the Controller transitions into the "Green" control state , with the "initialize" transition firing.

In the meantime, the EET of the "Facility" component displays the messages which are sent or received by the two "Merge" and "Controller" subcomponents. The broken horizontal lines separate the respective transitions of the automat. In the case of the "initialize" transition, for example, the Controller assigns the "Off" signal to the "Ind" output channel, the "Green" signal to "TL", and "Stop" to channel "PL."

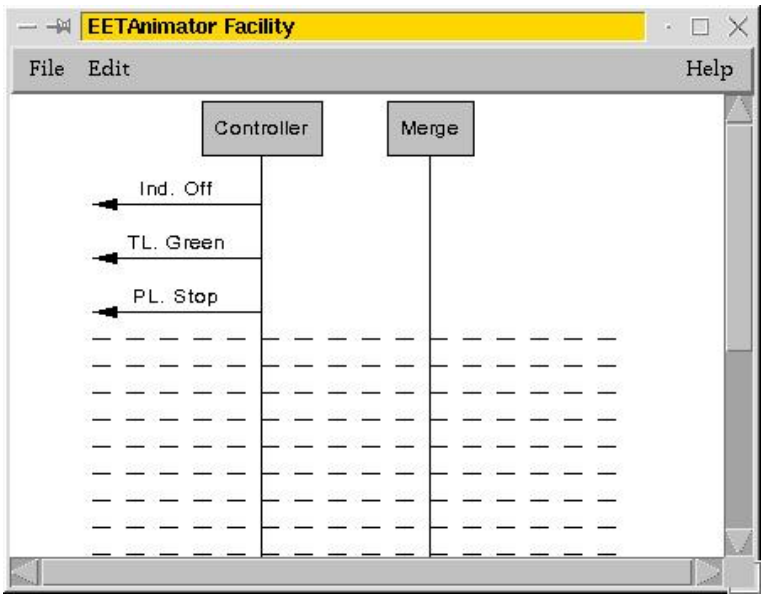

Figure 8.12: Start of the simulation in the EET of the "Facility" component

In the following steps, nothing seems to happen anymore. Indeed, no transition can fire in the "Green" control state of the Controller in the current situation. This is exactly the behavior we expect from a set of traffic lights: as long as no green phase for pedestrians is requested, the light for cars remains "Green" and the signal for pedestrians continues to show "Stop."

8.2.6 External Input

To make the further course of events more clearly observable, we bring the simulation to a halt using the ”Stop” button in the SimCenter (see Figure 8.2.4) window and, from now on, click the ”Single Step” button for each step.

Since the signal from ButtonA first reaches ”Merge” in accordance with our refinement of ”Facility” into the ”Merge” and ”Controller” components, we have a look at this component’s STD as well. For this purpose, we select ”Windows” -> ”STD Animators” -> ”Merge” in the window.

To request a green phase for pedestrians now, we enter ”Present” in the line below ”New value” in the ”Default Environment” window (see Figure 8.11). We finish our entry with the ”Enter” key.

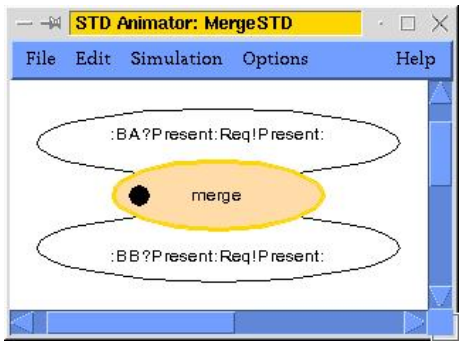

Figure 8.13: STD of the ”Merge” component

In the following animation step (click on the ”Single Step” button), the ”Present” signal reaches the ”Merge” component. After that (following more clicks) it is transmitted to the Controller through channel ”Request.” In the third step, the ”Receive request” transition fires in the STD of the Controller. In the EET we can watch the Controller binding its output channels to new signals.

The next time we click ”Single Step”, the ”Signal requested” transition fires and the values on the Controller’s output ports are updated in the ”Default Environment” window. After the next ”Single Step”, no change can be observed in any of our views. Therefore, we have a look at the local variable of the Controller.

8.2.7 Observing Local Variables

For this purpose, select ”Show Variables” from the ”Simulation” menu in the ”STD Animator: Controller” window. The following window is displayed:

Figure 8.14: Local variable t of the Controller

During the next eight steps, the variable t is gradually counted down. After `t` has reached the value `0`, the ”Switch to yellow” transition fires, and we see in the EET that the ”PL” and ”TL” output ports are bound to new values.

To observe the rest of the traffic lights cycle, we return to the automatic mode by clicking the ”Start” button. Finally, we reach the ”Green” state and thus the next green phase. By confirming ”Present” on ”BA” or ”BB” in the ”Default Environment” window again, we trigger another traffic lights cycle.

8.2.8 Animating SSDs

In the new cycle, we additionally observe the system structure diagram (SSD) of ”Facility.” For this, select ”Windows” -> ”SSD Animators” -> ”Facility” in the window.

Again, channels and ports are highlighted in color which data are currently bounded to.

By clicking ”Stop” in the window we finish the current simulation run. It is also possible to set the simulation back to the initial state by clicking ”Reset” from the ”Simulation” menu. ”Quit” closes all simulation windows. Before quitting, asks whether to save the generated EETs in the repository. If you answer yes, you’ll find the new EET under ”EET” in the project browser.

8.2.9 Simulation Recording

In addition to logging a simulation in EETs, SimCenter also records the communication of the system with its environment in the right half of the operating window. This is done using textual test data (TDFs):

With ”Save Test Data” from the ”File” menu, the test data can be saved in a file. Save your test data, for instance in a file called ”ampel.tdf.”

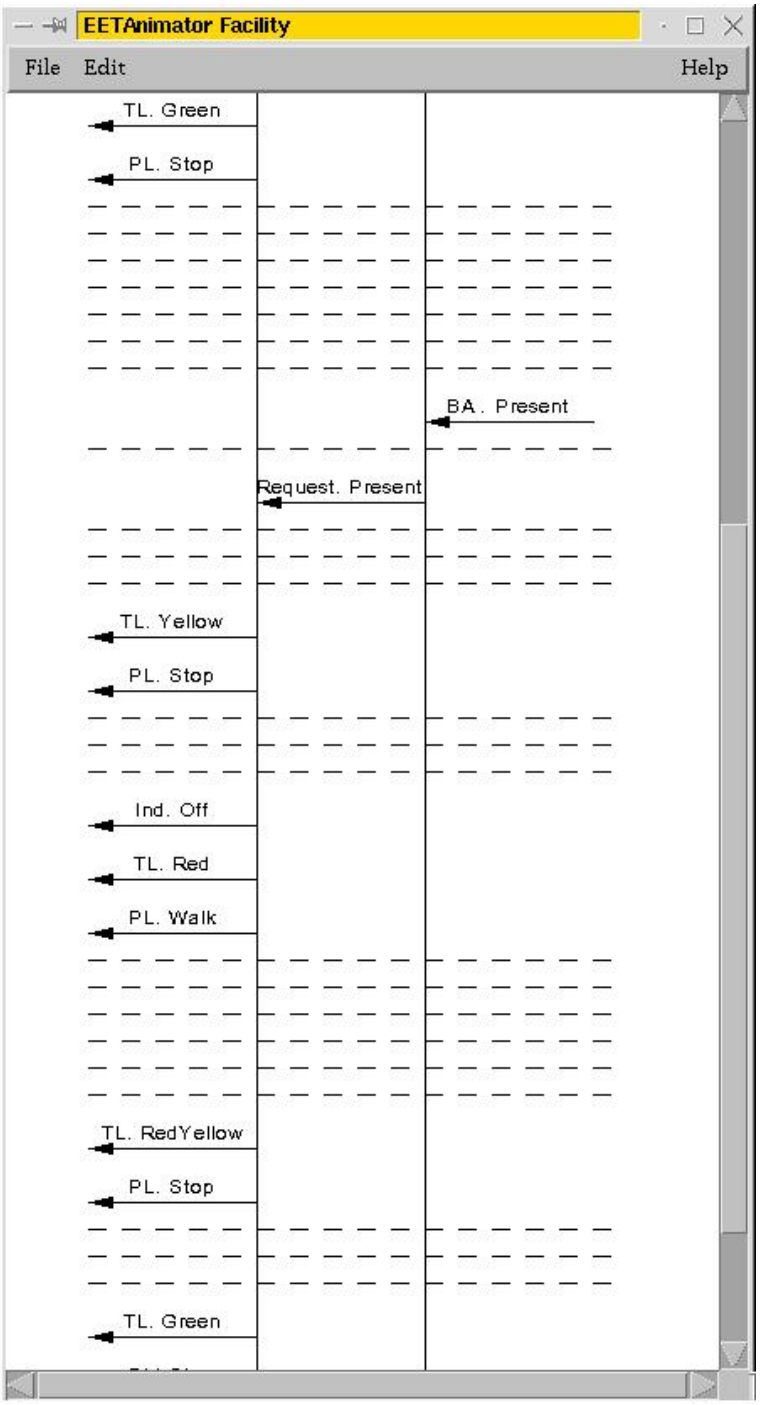

Figure 8.15: A complete traffic lights cycle in the EET

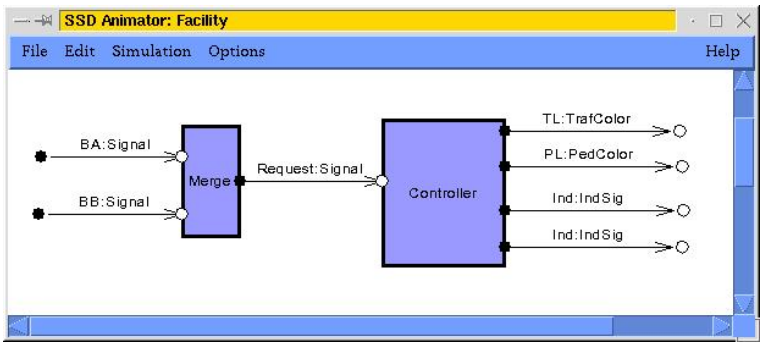

Figure 8.16: Animation of the SSD of the ”Facility” component

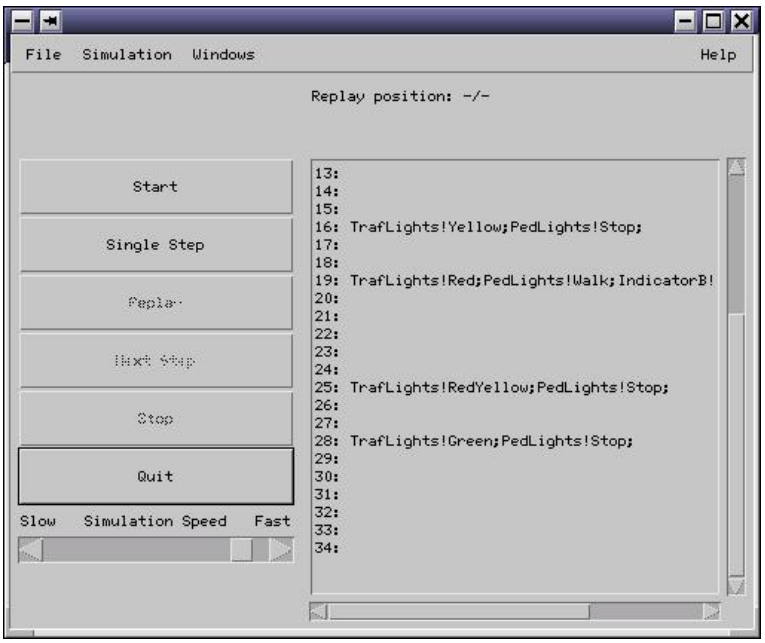

Figure 8.17: Recording of the traffic lights simulation

Now we can repeat the simulation run. Set the system back to its initial state using ”Reset” from the ”Simulation” menu of the operating window and reload your test data from ”ampel.tdf” with ”Load Test Data” from the ”File” menu.

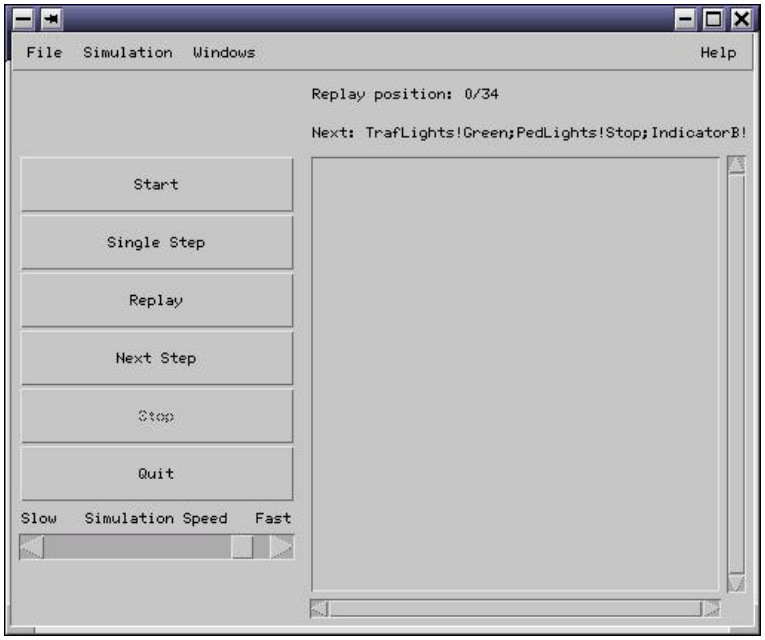

Figure 8.18: Traffic lights simulation before the replay

On the top right of the operating window, the number of steps contained in the test data and the current position is displayed. Now click ”Next Step” in the operating window to execute the next step from the test data, or ”Replay” to replay all test data. Click ”Stop” if you want to bring the replay to a halt.

You can alter or supplement the test data in the text editor of your choice. If you do this, it may of course happen that the file does not reflect a legal run of the system. In this case, SimCenter issues a warning and aborts the replay.

With ”Quit” from the ”Simulation” menu, all simulation windows are closed. Before quitting, AutoFOCUS asks whether to save the generated EETs in the repository. If you answer yes, you’ll find the new EET under ”EET” in the project browser.

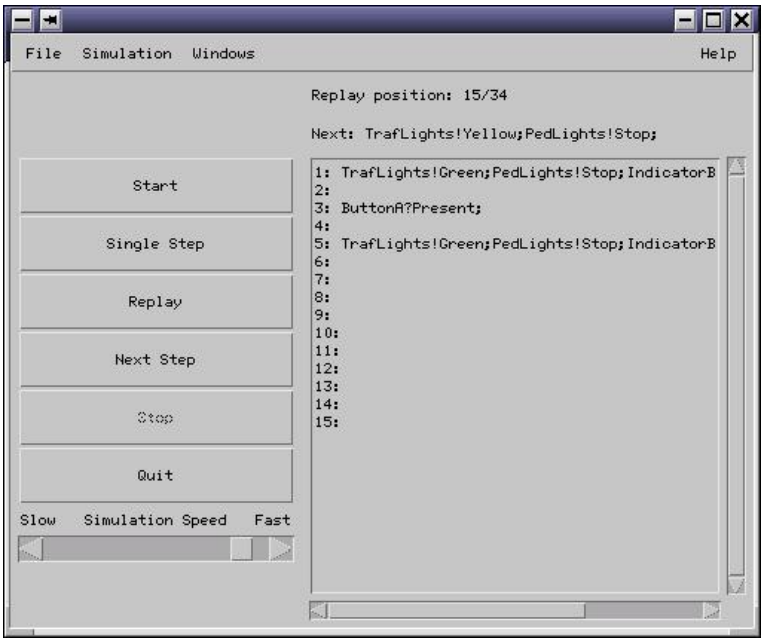

Figure 8.19: Replay of the recording

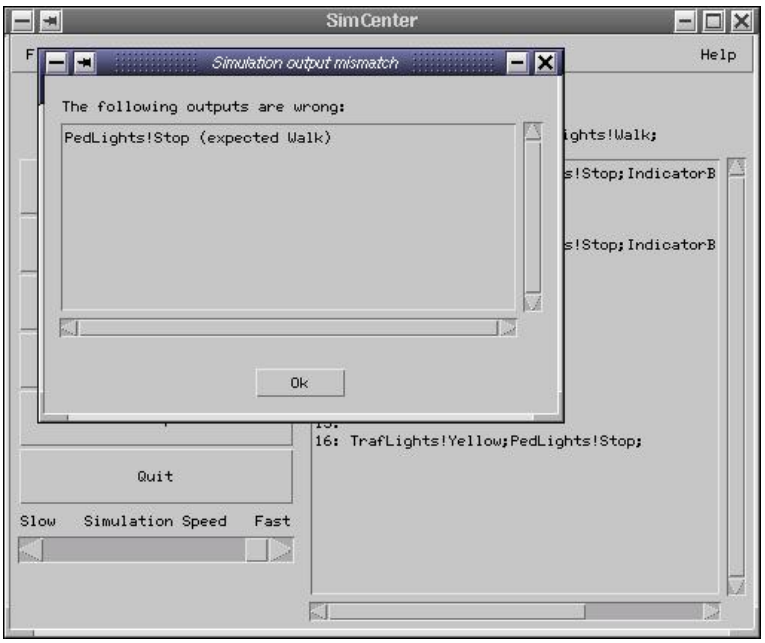

Figure 8.20: Abortion of the simulation

Chapter 9

Procedure

In the previous chapters, a traffic lights controller, which is a comparatively simple and manageable system, was modeled step by step. On the basis of this example, you became systematically acquainted with the terms and methods required to understand modeling in computer science, and learned how to apply them. The same modeling techniques, or at least similar ones, are used in practice even for large development projects spanning a long period of time and involving numerous people. Examples are the design of new processors, the development of complex application software, or the construction of space probes.When such projects are developed, not every little detail of the finished product is settled right from the start. Instead, it is necessary to make and possibly revise decisions in many phases. If errors are only recognized at the end of the development or, even worse, only when the allegedly complete system is already in use, corrections are difficult, expensive or even impossible to make.Therefore, it is obvious that such projects have to be handled in a stepwise, systematic fashion in accordance with standardized, comprehensible methods. In the following, we will briefly describe the typical phases. We will, however, deliberately not explain each term and each method again. Instead, the intention is to give you an overview of the process and the methods used.If complex systems (software and hardware) are developed, it is important to choose suitable description techniques. Using standardized models instead of documents and programs has numerous advantages. Models are

- more precise than language,
- more abstract than programs,
- clearer because of different views,
- applicable universally and in many phases,
- independent of programming languages

- easy to understand due to the graphical representation.

The modeling process describes how the models are created and refined. The Unified Modeling Language (UML) and the UML-RT Real-Time extension provide a large number of graphical notations which can be used for modeling. AutoFOCUS supports a semantically well-founded part of UML-RT.Just like all software development processes, the model-based development process consists of several development phases:

- planning (see Section 9.1)
- analysis (see Section 9.2)
- rough design (see Section 9.3)
- fine design (see Section 9.4)
- implementation (see Section 9.5)
- testing (see Section 9.6)
- maintenance (see Section 9.7)

When developing high-quality software, for example, it makes sense to include validation steps (see Section 9.8) between the various phases in order to eliminate errors in the systems already in the development phase. Universal (integrated) models and a simple and clear semantics (i.e. the meaning of the notation used) facilitate a precise (and tool-supported) validation. For this reason, AutoFOCUS uses only some of the models available in UML. For these models there is a simple semantics, which subsequently offers more validation options.The order of the various phases and the individual steps to be carried out are determined in the procedural model. Well-known models are the waterfall, V, or spiral models. Numerous procedural models require a top-down approach, which means that the system is first described as a whole and split up into components afterwards. If a bottom-up approach is used, the system is assembled from already existing components. In the design methodology (see Chapter 1) section, a part of the process was described, together with the description techniques used. The individual phases, as well as quality management, will be described in the following

9.1 Planning

In the planning phase, the development process is marked out. This involves the following technical decisions:

- determining the goals (of the system, safety analysis, development guidelines, structuring, documentation, quality certification, ...),

- selecting the models and description techniques,
- selecting the methods of development and validation (simulation, review, verification, test generation, tracing, ...),
- deciding on the tools.

Organizational aspects, such as planning resources, scheduling, and budgeting, are not part of the technical development planning.

9.2 Analysis

The analysis phase of the development starts from arbitrary (unstructured) requirements. Which methods were used to establish these requirements is irrelevant. Customer surveys, market analyses or interviews are some of the possibilities.At the beginning of the analysis phase, the requirements have to be structured. They are divided into requirements on the system and requirements on the environment. Depending on the kind of requirements, it may already be possible to draw up a somewhat more detailed structuring of the system, showing some of its subsystems, or to specify interfaces between the system and the environment and/or between subsystems. The result of the analysis is a requirements specification which describes the functionality of the system (the ”what” but not the ”how”). Models in the analysis phase must make it possible to distinguish between requirements on the system and the environment, and to describe interfaces. Moreover, it should be possible to use these models in later development phases as well.Use cases and sequence diagrams describing the input and output behavior of the systems in time sequences are especially suitable for specifying the functionality of reactive systems. In AutoFOCUS, system structure diagrams and sequence diagrams are used to model the requirements.Often there are requirements which necessitate a somewhat more detailed structuring. For instance, a safety analysis must refer to the rough system architecture. In this case, there is no clear-cut dividing line between the analysis and rough design phases. There are also requirements on the fine design or the implementation, for example that a specific programming language be used.The methods employed in the analysis phase depend on the modeling goals. For example, there are special methods of looking for and/or preventing vulnerabilities in a safety analysis.

9.3 Rough Design

In the rough design phase, the architecture and the behavior of the system to be developed are determined. Here, the experience of the developers is called for.

The models used in the design must represent the design decisions in a clear and readable fashion and should match the models in the analysis phase.In AutoFOCUS, system structure diagrams and sequence diagrams are used in the rough design phase as well, but the level of detail is now much finer than before. When the rough design is completed, all components in the system are described with their interfaces and interactions. From a detailed design, an executable simulation can be generated.

9.4 Fine Design

In the fine design phase, the design decisions are translated into constructive models. This means that for all components a behavior regarding their interfaces is being defined. The interactions specified in the rough design phase serve as patterns for these models.For modeling reactive systems, state-based models (automata) have proved their worth. In AutoFOCUS, state transition diagrams are used.

9.5 Implementation

In the implementation phase, the advantages of a model-based development become obvious, because programs can now automatically be generated from the detailed, constructive models. The programming language - with the corresponding code generators - can be freely selected. Components which were not modeled in sufficient detail have to be implemented manually. This applies also to special interfaces (e.g. to the hardware).

9.6 Testing

As far as simple, basic functions are concerned, functional tests will do, i.e. tests to find out whether certain input values yield the expected outputs. In state-based, reactive systems, however, these functional test cases are no longer sufficient, because the outputs of the systems depend on the current state. To be able to test the system in different states, we must carry out test sequences. Test sequences can be

- sequences from the requirements specification or the fine design,
- executions of prototypes (simulation) or

- sequences generated (from the models) to meet special testing goals.

While there are many different methods (”testing strategies”) of generating test sequences from models, these sequences cannot point out errors in the program code generated from the models. However, testing can reveal errors if parts of the program code were not automatically generated or if hardware components are used in the real world which were not specified correctly in the model. Many quality standards require careful tests, during which coverage is measured and controlled in the executed programs (i.e. it is checked which statements were executed in which way).

9.7 Maintenance

The maintenance of a system begins after its completion. In this phase it is particularly important to integrate changes requested by the customer. A precondition for this is a well-documented and comprehensible development. In addition, one must make sure that if any requirements are changed, all affected parts of the model are revised and the necessary development steps are repeated.

9.8 Quality Assurance

Quality assurance makes great demands on the development process and is part of all development phases. However, different techniques are employed at each stage. Important quality assurance procedures are

- review,
- safety analyses,
- simulation,
- consistency checks,
- tracing,
- testing,
- proofs.

These procedures can be divided into validation techniques and verification techniques. During validation, the specifications, models, and programs are tested for plausibility, for instance by means of reviews and simulation. During verification, steps are carried out which must be unambiguous for the machine and

repeatable, such as consistency checks, tests, and formal proofs.In different development phases, different quality assurance procedures are employed. Important procedures in the development phases include:

- analysis: weak-point analysis, attacker modeling,
- rough design: risk analysis, simulation of prototypes,
- fine design: simulation, model checking (efficient and complete testing),
- implementation: testing (with generated test cases).

Other techniques (review, safety analysis, tracing, consistency checks, and proofs) can be applied in all phases. Tracing means that the requirements are tracked throughout the development process to ensure that none of them are overlooked.Consistency checks are a particularly important technique. They are based on the connections between the various views and the redundancy of the description techniques and can be performed automatically. In UML, consistency checks can be formulated using the OCL language. Extensible consistency checks are integrated in AutoFOCUS.In a model-based development process, the following quality assurance steps are required, among others

- During the analysis stage, quality assurance consists in checking whether the informal requirements were realized in the models (tracing). For example, each requirement should be assigned either to the system or to the environment to ensure that no requirement was overlooked.
- During the rough design phase, one has to make sure that the interactions and the interfaces of the system match. For example, no message can leave the system if there is no suitable port in the model of the interface.
- During the fine design stage, simulation can be used to check whether the system carries out the interactions laid down in the analysis and design phases.

Thanks to model-basing, a number of quality assurance procedures can thus be performed automatically even during the development process