

# Quest Users Guide

Peter Braun, Heiko Lötzbeyer, Oscar Slotosch

Version 1.0 of March 7, 2000

## Abstract

Within the project Quest several tools have been connected. The central tool is AUTOFOCUS, the connected tools are VSE, SMV, CTE, and SATO. Furthermore several parts (a temporal specification language, a selection tool, a test driver, and an abstraction chooser) have been added to improve the integration of the tools and to support the development of correct software. Furthermore we describe some examples that have been used to test the quest tools.

## Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Conventions used in this manual . . . . .	7
<b>2</b>	<b>Views and Models of AUTOFOCUS</b>	<b>8</b>
2.1	Structure of Views in AUTOFOCUS . . . . .	8
2.2	Structure of the Models . . . . .	9
<b>3</b>	<b>The Language QuestF: DTDs and Properties</b>	<b>11</b>
3.1	Type Definitions . . . . .	12
3.2	Terms . . . . .	14
3.3	Function Definitions . . . . .	14
3.4	Module Definitions . . . . .	15
3.5	Properties . . . . .	16

3.6	Correctness Conditions . . . . .	18
3.7	Integration and Application . . . . .	18
3.8	Predefined Elements . . . . .	18
3.8.1	Types . . . . .	18
3.8.2	Operations . . . . .	19
<b>4</b>	<b>Using the Model Browser</b>	<b>20</b>
4.1	Starting the browser . . . . .	20
4.2	Working with repositories . . . . .	20
4.3	Viewing and editing nodes . . . . .	21
4.4	Creating new nodes . . . . .	21
4.5	Deleting nodes . . . . .	22
<b>5</b>	<b>Integration</b>	<b>22</b>
5.1	Integration in AUTOFOCUS . . . . .	22
5.1.1	Exporting Projects from AUTOFOCUS . . . . .	22
5.1.2	Importing Projects in AUTOFOCUS . . . . .	22
5.1.3	Limitations . . . . .	22
5.2	Integration in VSE . . . . .	23
5.2.1	Overview . . . . .	23
5.2.2	Importing Projects in VSE . . . . .	24
5.2.3	Exporting Projects from VSE . . . . .	28
5.2.4	Limitations . . . . .	31
<b>6</b>	<b>Model Checking with AUTOFOCUS</b>	<b>31</b>
6.1	Introduction . . . . .	31
6.2	Counter Examples . . . . .	33
6.3	Connected Checkers . . . . .	33
6.3.1	Using SMV . . . . .	33
6.3.2	Using SATO . . . . .	35
<b>7</b>	<b>Abstraction Chooser</b>	<b>35</b>

7.1	Methodology . . . . .	35
7.2	Abstraction Example: Comparator . . . . .	38
7.3	Using the Abstraction Chooser . . . . .	40
<b>8</b>	<b>Connection to VSE</b>	<b>45</b>
8.1	Introduction . . . . .	45
8.2	Translation from AUTOFOCUS to VSE . . . . .	47
8.2.1	SSD . . . . .	47
8.2.2	STD . . . . .	59
8.2.3	DTD . . . . .	66
8.3	Translation from VSE to AUTOFOCUS . . . . .	71
8.3.1	SSD . . . . .	71
8.3.2	STD . . . . .	72
8.3.3	DTD . . . . .	72
<b>9</b>	<b>The Test Environment</b>	<b>73</b>
9.1	Introduction . . . . .	73
9.2	Creating Test Sequences . . . . .	74
9.2.1	Test Basis: SSDs . . . . .	74
9.2.2	Test Basis: SSD with STDs . . . . .	76
9.2.3	Test Basis: Functions . . . . .	77
9.2.4	Deriving sequences from test automata . . . . .	77
9.2.5	Test Basis: EETs . . . . .	78
9.3	Performing Tests . . . . .	78
9.3.1	The Test Object Interface . . . . .	78
9.3.2	Running the Test . . . . .	78
9.3.3	Interpreting the Results . . . . .	79
<b>A</b>	<b>Installation</b>	<b>79</b>
A.1	System Requirements . . . . .	79
A.2	Installation . . . . .	81

A.3	Configuration	84
A.4	Uninstallation	84
<b>B</b>	<b>Case Studies and Examples</b>	<b>84</b>
B.1	Emergency Closing System of Storm Surge Barrier	85
B.1.1	The Original Model	86
B.1.2	An Improved Model	87
B.1.3	A Comparator for Sensor Signals	87
B.2	FM99 Banking System	87
B.2.1	Complete Model	88
B.2.2	Abstract Model	88
B.3	Traffic Light Control System	88
B.4	Other Work	89
<b>C</b>	<b>Correctness of Abstractions</b>	<b>90</b>
C.1	Definition of the Abstraction Function	90
C.2	Homomorphosm Proof Obligation	91
C.3	Strengthening Proof Obligation	93

# 1 Introduction

In this user guide we describe the tools developed within the project Quest. The goal of the project Quest was to combine graphical software development tools with tool to ensure the quality of the developed software. There are different kinds of such tools:

- **model checker:** can be used to check temporal properties of finite models automatically.
- **theorem prover:** support manual verification of arbitrary models and properties
- **test tools:** test the developed software

During the analysis phase of the project we selected AUTOFOCUS [HMR<sup>+</sup>98, HMS<sup>+</sup>98] for graphical system specifications, and decided to connect the model checkers SMV [RSW97] and SATO [Zha97], and the theorem prover VSE II to AUTOFOCUS. For testing we selected the CTE [GWG95] for the classification of variables and implemented several methods to generate test cases from AUTOFOCUS specifications. Furthermore a test driver has been implemented to test Java programs against the test cases.

Figure 1 shows the structure of the tools of the project Quest. The boxes represent the connected tools, and the arrows between them denote the connections that are designed and implemented within the project Quest. We integrate the tools by defining interfaces and (sometimes partial) translations between the used concepts. The interfaces use the “QML” (Quest model language) model format, and can also be used by other tools. All tools are connected to the models of AUTOFOCUS. The models are exported from AUTOFOCUS, that is be used to specify the model using graphical views. The models are represented in a textual format (QML), and can be managed using the model browser. Furthermore many quest programs can be started from this model browser.

For the connections of AUTOFOCUS to the other tools two new features have been integrated into AUTOFOCUS:

- the functional language QuestF, and
- the textual interface (export and import) QML.

The functional language QuestF is used in DTDs, transitions in STDs, and in properties. It is the logical basis for the translations to the other tools. The interface is used to import and export projects from AUTOFOCUS to the model browser.

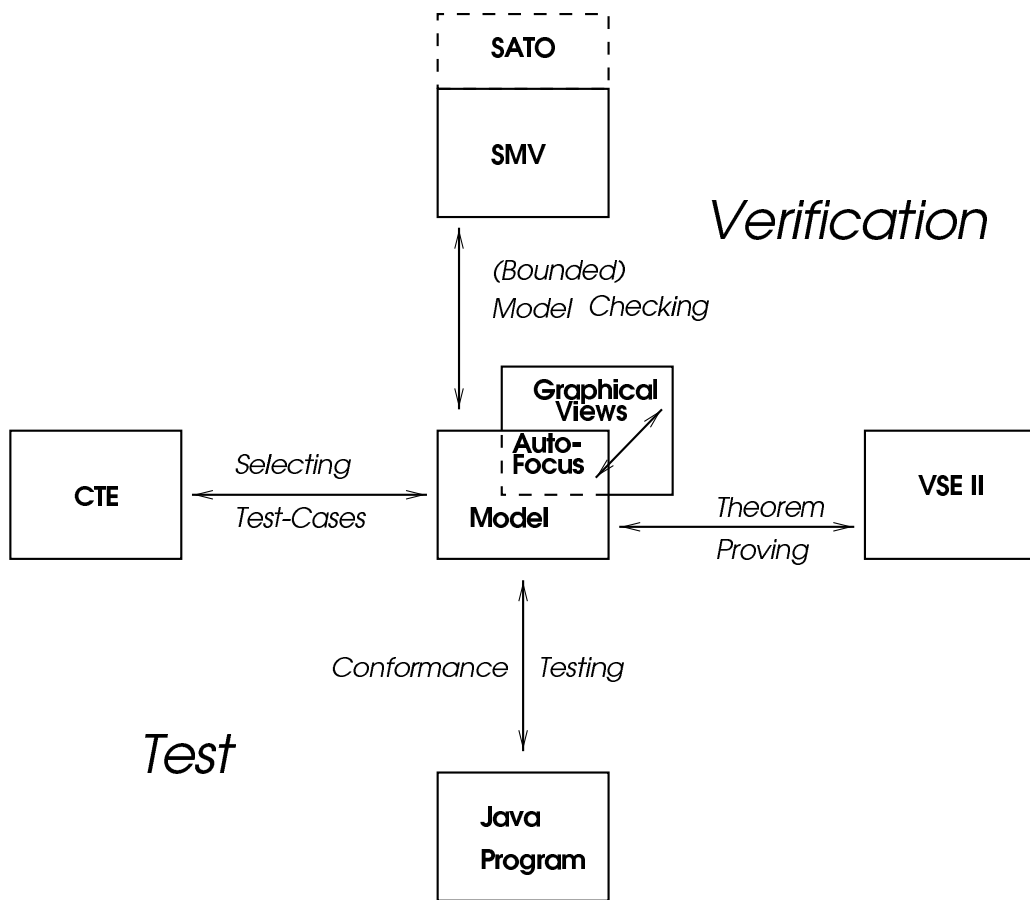


Figure 1: Tool Structure of Quest

This user manual is a description of the Quest extensions of AUTOFOCUS. We assume the user to be familiar with AUTOFOCUS models and terms like “STD”, or “type”. This manual describes the features implemented within the project Quest, and the user interface to use them. There are several publications (available on the Quest homepage<sup>1</sup>) that describe more general aspects.

- Quest: Overview over the project Quest [Slo98],
- Process: Enriching the Software Development Process by Formal Methods [BS99],
- Integration concept: Consistent Integration of Formal Methods [BLSS00],
- Model checking design: The Quest for Correct Systems: Model Checking of Diagrams and Datatypes [PS99],

This user manual is structured as follows: In Section 2 we present the document and model structures within AUTOFOCUS including the extensions like properties, that are not part of AUTOFOCUS. In Section 3 we present the language QuestF by introducing and explaining examples for types, functions, and properties. Section 4 contains a description of the user interface of the model-Browser, and Section 5 describes how the browser is integrated into AUTOFOCUS and into VSE. The translation of models to SMV is described in Section 6, and Section 7 explains the usage abstractions to extend the applicability of model checking to larger and infinite systems.

Section 8 describes the connection to the theorem proving environment VSE. The test environment and the methods to generate specification based test cases are explained in Section 9.

The appendices A and B conclude the manual with an installation guide and an overview over the examples shipped with the software.

## 1.1 Conventions used in this manual

This manual uses different fonts to represent different types of information.

Shell commands appear in a monospaced font. For example:

```
command [-optional-argument] <filename>
```

---

<sup>1</sup>[www4.in.tum.de/proj/quest](http://www4.in.tum.de/proj/quest)

In this example you have to type `command` in your command-shell and you can give it an `-optional-argument` and you have to specify a filename.

Menus also appear in a monospaced font. For example: `File`

Buttons are presented like this: `OK`, `Cancel`

## 2 Views and Models of AUTOFOCUS

AUTOFOCUS has a view oriented structure, whereas the extensions of the project Quest use the model (semantics). The main model of AUTOFOCUS are components. Within the model browser there are some additional attributes available, that cannot be represented graphically within the views of AUTOFOCUS.

Exporting a project from AUTOFOCUS into the QML generates a model for the views and requires that the views are consistent, for example each component in a SSD should have a refinement or a behaviour. This can be checked using the consistency test of AUTOFOCUS (see [HSE97]).

Since the model browser and the browsers of AUTOFOCUS represent the structures of models and views, we use them to present the structures.

### 2.1 Structure of Views in AUTOFOCUS

AUTOFOCUS has a view oriented structure (see for example [HMR<sup>+</sup>98, HMS<sup>+</sup>98]). Every document describes a graphical view on the model. The main structure of the views are the kinds they belong to. There are SSDs, STDs, EETs, and DTDs in the browser of AUTOFOCUS (see Figure 2).

Since the views in AUTOFOCUS are hierarchic, within each document class there can be substructures. Components in SSDs can be refined with SSDs, states in STDs can be refined by STDs, and components or boxes in EETs can be refined using EETs.

DTDs have an inclusion structure, since each DTD can import other DTDs. This leads to a graph structure instead of a tree structure, however this is not yet implemented within AUTOFOCUS. Copy and paste has to be used to reuse DTDs.

The association that relate views of different kinds cannot be visualized in the browser of AUTOFOCUS Atomic components cannot be seen in the browser, since it shows only the available documents. The only visible association in the browser is the substructure association between views of the same kind. Further associations, for example the assignment of a STD to a component cannot be seen in the



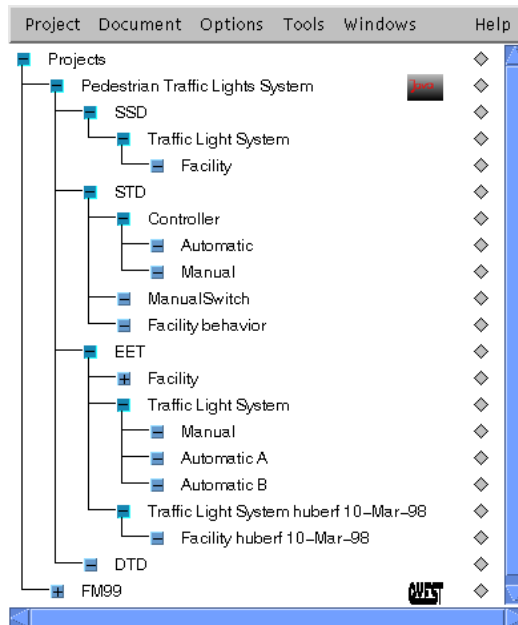


Figure 2: The Browser of AUTOFOCUS

browser of AUTOFOCUS.

## 2.2 Structure of the Models

QML has a model oriented structure. The main models are components. The model browser shows more details and attributes of the model than AUTOFOCUS does, and in addition there are some attributes that are not present in AUTOFOCUS, for example properties, abstractions, or an implementing code.

Every component can have subcomponents. This reflects the hierarchy in AUTOFOCUS SSDs. In contrast to the browser of AUTOFOCUS the STDs assigned to a component are also displayed (see Fig. 3). In the model browser the following model/structures are displayed:

- **Component:** with attributes
  - Subcomponents
  - Ports
  - Automaton (STD)
  - EET

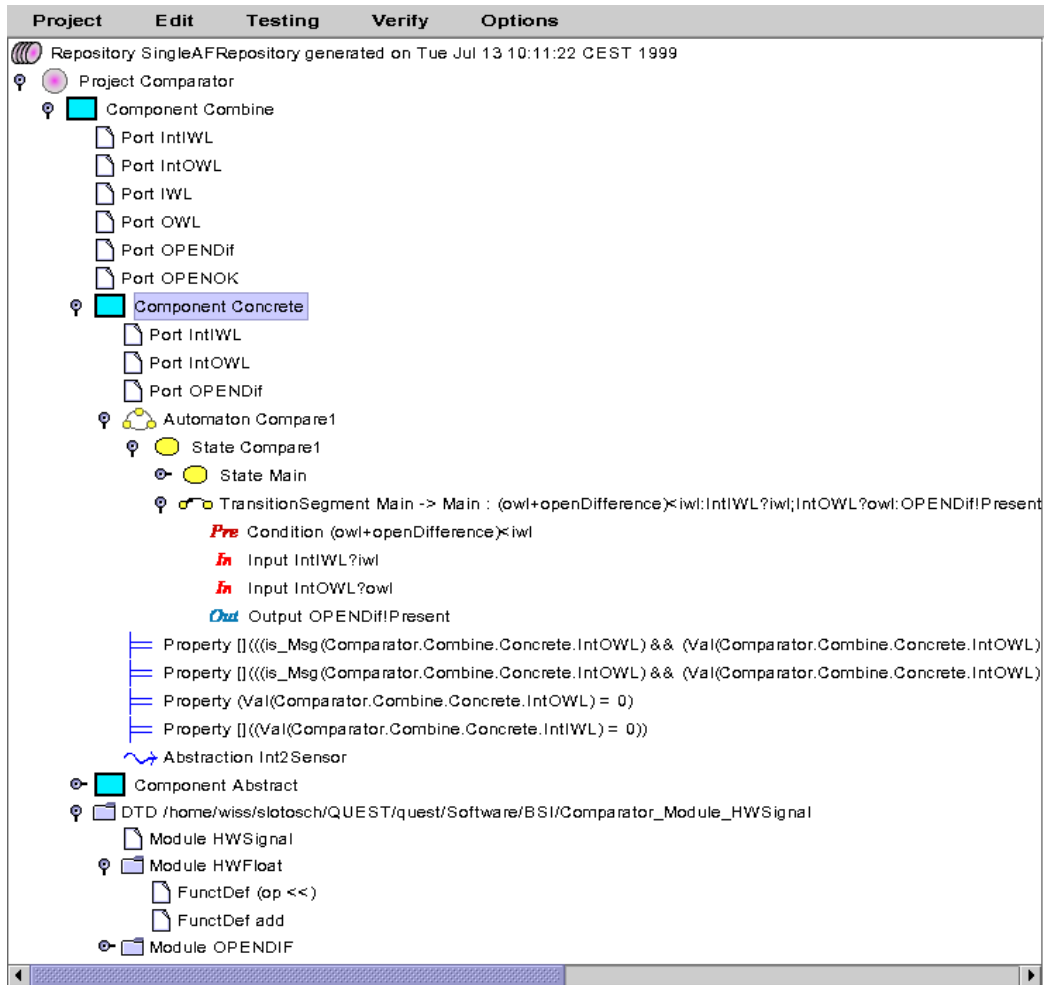


Figure 3: The Model Browser

- DTD
- Property
- Abstraction
- **Automaton (STD)**: with an attribute state for the description of the behaviour
- **State**: with attributes
  - Substates
  - transition segments (parts of transitions)<sup>2</sup>
- **Transition Segment**: with attributes
  - Precondition
  - Input patterns
  - Output patterns
  - Actions
- **EET**: with attributes
- **DTD**: with several modules
- **Modules**: related types and functions

See Section 4 for a description of the features of the model browser.

### 3 The Language QuestF: DTDs and Properties

In this section we describe the language QuestF. QuestF is a functional language, extended with some constructs for the description of system properties.

The functional part of QuestF is similar to ML [Pau91] or Gofer [Jon93], the temporal operations are like those in TLS [Mül98a], a subset of LTL. The operations for the system descriptions are close to the notations used in the AUTOFOCUS tool.

---

<sup>2</sup>The model supports the concatenation of different transition segments to one transition (Inter level transitions). In AUTOFOCUS inter level transitions (between documents) require identical labels/semantics, otherwise they cannot be unified during the simulation or the QML export.

In the following subsections we provide syntax for the language QuestF, and explain it by some examples of a banking system (see Appendix B.2 or <http://www4.in.tum.de/proj/quest/> for a description of the example). Furthermore we characterize the subsets of QuestF, that can be used for model checking, and for the export to VSE.

### 3.1 Type Definitions

Type definitions are the central part in the DTDs of AUTOFOCUS/Quest. They define types and elements of QuestF. For example

```
data Message = Money(Int) | NoMoney | Balance | MailSent;
```

defines the type `Message` and the **elements** `NoMoney`, `Balance`, and `MailSent`, and the function `Money: Int -> Message` that constructs an element of type `Message` for each argument of type `Int`. The elements and functions defined within the `data` construct are called **constructors**, and can be used in **patterns** to define functions or transitions with **pattern matching** (see Section 3.3).

The defined types can be used in SSDs together with the default types `Bool`, `Int`, `Float`. The defined elements can be used in the transitions of STDs, for example in

```
m <= 0 ; x ? Money(m); s ! m, ans ! NoMoney; Msg = Money(m)
```

This transition reads from the channel `x` of type `Message`, and writes to the channels `s` of type `Int` and `ack` of type `Message`. The transition can only be executed, if the following conditions hold:

- a value is on the channel connected to the port `m`,
- the value matches the pattern `Money(m)`, and
- the recondition `m <= 0` holds.

For example the transition cannot be executed if `x` has no value, or `MailSent`, or `Money(100)`. In the case `x` has the value `Money(0)`, the transition can be executed and sends the value `0`<sup>3</sup> to the port `s`.

Type definitions can also be used to define `polymorphic` types, for example lists, by:

---

<sup>3</sup>The transition variable `m` is instantiated to the value `0` during the pattern matching of the conditions.

```
data List(a) = Nil | Cons(first:a,rest:List(a));
```

In this definition two **selector functions** are defined: `first: List(a) -> a` and `rest:List(a) -> List(a)`. Both are partial functions and can only be applied to list elements constructed with `Cons`. Therefore type definitions introduce predicates (discriminator functions) `is_Nil` and `is_Cons`. The semantics of discriminator and selector functions are (see Section 3.3) for function definitions).

```
// generated selector functions:
fun first(Cons(x1,x2)) = x1;
fun rest(Cons(x1,x2)) = x2;
// generated discriminator functions:
fun is_Nil(Nil)=True
  | is_Nil(x) = False;
fun is_Cons(Cons(x1,x2))=True
  | is_Cons(x) = False;
```

For model checking finite types are required. Therefore the type definitions must not be recursive, i.e. the defined type must not occur in the definition. For example the above type `List` is not finite. Furthermore polymorphic types cannot be used for model checking<sup>4</sup>.

The grammar for data definitions (*ddata*) is (in BNF-Syntax):

```
ddata ::= data string tvars? ≡ dconstr dconstrs* ;
tvars ::= ( tvar cotvar* )
cotvar ::= , tvar
tvar ::= string
dconstrs ::= | dconstr
dconstr ::= string
              | id ( karg kargs* )
              | ( karg infid karg )
kargs ::= , karg
karg ::= type
          | id : type
```

Type definitions can be used in DTDs to define additional functions (see Section 3.3).

---

<sup>4</sup>The only exception is the predefined polymorphic type `Channel(m)`, see Section 3.8.

## 3.2 Terms

In the language QuestF the following terms are allowed:

- atoms (constants, variables), like 2, or X
- applications like `not(x)` or `mult(x,y)`
- infix-applications like `2+3` or `(x ^ y)`
- conditionals like `if x then y else z fi`, where the else branch can be omitted, and
- unnamed functions (“lambda abstractions”) like `param x y { t }`.

Around arbitrary infix operators there are brackets required. Predefined infix operators (see Section 3.8.2) have priorities, that allow to write terms like `a+b+c*d`.

The important grammar rules for building terms are:

```
cterm ::= cbase typean?  
typean ::= ; type  
cbase ::= sign? id  
          | param string+ { cterm }  
          | ( cterm infix cterm )  
          | if cterm then cterm else cterm fi  
          | if cterm then cterm fi  
          | unop ( cterm )  
          | ( cterm )  
sign ::= ±  
        | =
```

Lambda abstraction is not supported for model checking and VSE generation. In the next section we show how to define functions in QuestF.

## 3.3 Function Definitions

In the language QuestF functions can be defined by pattern matching (like in functional languages). In the example of the banking system (see Appendix B.2), we have the following example:

```

data Card = Invalid
          | Valid(info:Cardinfo,dayAmount:Int,Date);
const maxAmount=2;
fun maxForToday(day,Valid(ci,amt,lastDay)) =
  if (lastDay == day)
  then maxAmount-amt
  else maxAmount
  fi;

```

It defines a constant `maxAmount` and a function `maxForToday`, using the constructor function `Valid` for pattern matching. Since the function `maxForToday` has no branch for invalid cards, it is undefined for this pattern.

Infix operations can also be defined, for example

```

fun (Nil # s) = s
  | (Cons(x,s) # t) = Cons(x,(s#t));

```

Defines a concatenation operator `#` for lists.

The grammar for constants and functions is:

$$\begin{aligned}
 \text{decl} ::= & \mathbf{const} \text{ id} \equiv \text{cterm} ; \\
 & | \mathbf{fun} \text{ cterm fundecls}^* ; \\
 \text{fundecls} ::= & \downarrow \text{cterm}
 \end{aligned}$$

For function definitions the form  $left = right$  is required (see Section 3.6). On the left side only variables and constructors are allowed. All variables used in the right part of the function definition have to occur in the left part. Every variable may occur only once within the left part of the equation.

### 3.4 Module Definitions

The QuestF languages has a modular structure. Every component (see Section 2) has a list of DTDs. A DTD is a module within the QuestF language, the name of the DTD is the name of the module. Every DTD has a tree of sub-DTDs, that correspond to imported modules. For each DTD (and the corresponding tree) the export into QuestF format generates a file with a module structure.

The user can create (imported) sub-DTDs using the “Navigate” menu of the DTD-Editor (see Figure 4), therefore it is not necessary to look at the grammar for modules in QuestF.

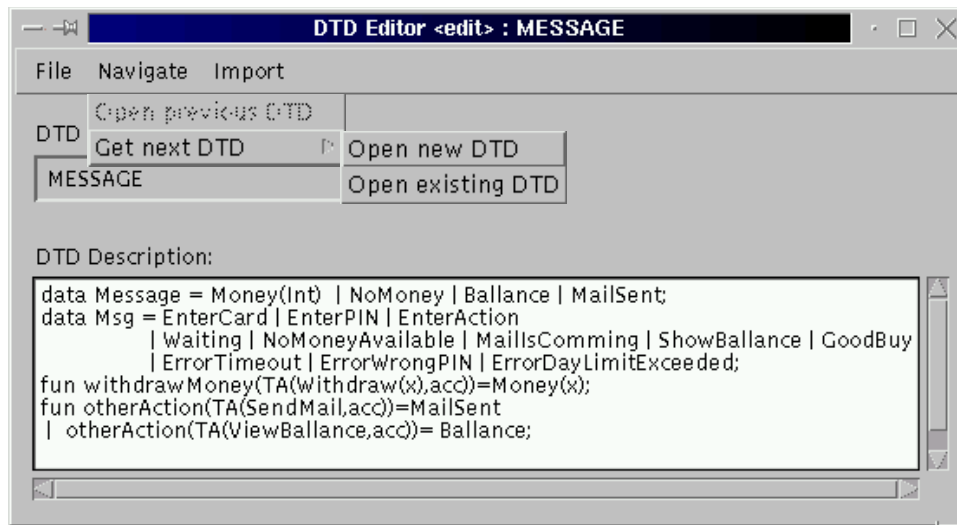


Figure 4: Importing a DTD Module

In the DTDs of AUTOFOCUS only tree structures are supported, whereas the QuestF language supports arbitrary imports.

### 3.5 Properties

The QuestF language allows to formulate properties. Properties can be defined for every component (see Section 4 for the user interface). Properties are temporal formulas over components. The following temporal operators are supported:

- $\square(f)$ :  $f$  is always true
- $\diamond(f)$ :  $f$  is true sometimes in the future
- $\circ(f)$ :  $f$  is true in the next state

Formulas can be connected with the boolean operators for terms (see Section 3.8.2).

Formulas can refer to attributes of the component, and to the operations available on these model attributes:

- `StateName`: refers to a state named `StateName`, `@` is the current state. On states the only operations are



- =: equality
- !=: inequality
- PortName: refers to a port named PortName. On ports the following operations can be applied:
  - p!x port p sends the value x.
  - p?x port p receives the value x. x can also be a pattern, consisting of variables and constructor functions. Furthermore all functions available on the type Channel (see Section 3.8.1) can be applied, for example to state that a port p is empty by `is_NoVal(p)`.
- LocVar: refers to a local variable LocVar of this component. All operations available on the variable can be applied.
- SubName: refers to a subcomponent named SubName. On subcomponents all attributes can be referred using a . as delimiter.

With the possibility to refer to subcomponents, we have a qualification mechanism for properties.

Properties are automatically translated into a fully qualified form after they have been entered.

An example for a property of the component `Connection1` is:

```
[ ](Answer?NoMoney => ()(CentralMsg!NoMoney))
```

It is expanded to the fully quantified property

```
[ ](FM99.BankingSystem.Connection1.Answer?NoMoney =>
    ()(FM99.BankingSystem.Connection1
        .CentralMsg!NoMoney))
```

Since properties are built like usual terms (over temporal operators and model selectors), there is no additional grammar required for properties.

Note: Since identifiers in QuestF, must not contain blanks, properties must not contains identifiers with blanks. Since qualified names contain the name of the project, and the name of all components, these names must not contain blanks as well, if qualified properties should be stored or loaded.

## 3.6 Correctness Conditions

The most important correctness condition for expressions in QuestF is type correctness. Furthermore input and output ports must not be confused. An additional source of errors are reference problems, for example to non-existing attributes of components, or due to wrong qualified names. Some of these errors are detected when the model is used, i.e. when the model is exported or translated. This causes exceptions that point to the invalid references.

## 3.7 Integration and Application

The QuestF languages cannot be type checked in the AUTOFOCUS tool. However when importing a model into the model browser type checks can be applied.

## 3.8 Predefined Elements

In this section we describe the predefined elements of QuestF.

### 3.8.1 Types

The following data types are predefined in QuestF<sup>5</sup>

- **Bool, boolean, Boolean** with **True, False** and **true, false**.
- **Int, int** with **0, -1, 1, ...**
- **Float, float** with **0.0, ...**
- **a -> b** for functions from type a to type b
- `Channel(m) = NoVal | Msg(Val:m);`

The type `Bool` can be translated to VSE and can be used for model checking. `Float` and higher order functions (like in ML) are not supported within SMV and VSE. The type `Int` is restricted to a finite domain for model checking, ranging from 0 to `MaxInt`, a value that can be defined in the model browser. The polymorphic type `Channel(m)` must not be used in AUTOFOCUS. In QuestF it is used within properties, for example to express `x?` by `is_NoVal(x)`. In the

---

<sup>5</sup>Allowing alternative syntactic forms for some types and elements makes it easier to convert projects with from other languages to QuestF format.

translations to SMV, SATO and VSE this type is used for a type correct representations of channels (see Sections 6, and 8).

### 3.8.2 Operations

In this section we describe predefined operations (and their priority), that can be used to build terms. Priorities range from 1 (low) to 4 (high), and can have an associativity, for example  $\&\& \ 2l$  denotes that  $\&\&$  has priority 2 and associates to the left.

The following operations are allowed on all terms:

- equality:  $= \ 1l$
- inequality:  $!= \ 3l$
- receive:  $? \ 3$
- send:  $! \ 3$

The following operations are allowed on boolean terms:

- negation: `not`
- implication:  $=> \ 1r$
- biimplication:  $<=> \ 1r$
- disjunction:  $|| \ 1l$
- conjunction:  $\&\& \ 2l$

The following operations are allowed on numeric (`Int` and `Float`) terms:

- subtraction:  $- \ 2l$
- addition:  $+ \ 2l$
- greater:  $> \ 3$
- greater or equal:  $>= \ 3$
- less:  $< \ 3$
- less or equal:  $<= \ 3$

- multiplication: \* 41
- division: / 41

Note that multiplication and division cannot be translated to SATO (see [Wim00]).

## 4 Using the Model Browser

The model browser of the Quest development environment is the starting point for all verification and testing activities. It enables you to

- load a repository
- view the repository
- edit the repository in a limited way and
- start the verification and testing tasks

### 4.1 Starting the browser

The browser does automatically come up when you start the Quest development environment. This is done by typing

```
startQuest
```

from a command shell. See Section A for further details on the installation.

### 4.2 Working with repositories

With the `Files` menu in the menubar you can open, close, save and import the repository.

**Open a repository:** There are two ways to open a repository. When opening a repository, an optional type check can be performed. If you want to leave out the type check use `open repository (fast)` to open the repository. After the repository is loaded the repository name and all projects will be displayed in the browser. If you load the repository with `open repository (check)` all terms in your model will also be type checked while the repository is loaded. Use this if you want to type check the model. Type checking the model, especially the transitions of STDs cannot be performed in AUTOFOCUS.

If you are currently working on a repository and open a new one, the current repository will be closed before the new one is opened. So be patient and save the contents of the current repository before opening a new one.

**Close repository:** Closes the repository and throws away all changes since last save.

**Save repository:** Save the repository to the same file where it was loaded from. In the current version it is not possible to save the repository to another file. Therefore the repositories should be copied to a new file before they are loaded into the browser. Beside the repository file you also have to copy all module files in order to read in the copied repository properly.

**Import repository:** Import all projects of the specified repository into the repository that is currently loaded into the browser. This function does not perform the type check.

### 4.3 Viewing and editing nodes

If you want to view the attributes of the selected elements, check the box in `Options->Show Attribute Window`. All nodes of the current selection will be displayed in a separate window. The attributes of some nodes can be edited in the attributes window (e.g. messages).

### 4.4 Creating new nodes

With the model browser new subnodes can be created. The current version does only support the creation of a Code subnode for a component. Future versions

will allow the creation of arbitrary nodes. In order to create a new Code subnode first select the associated component node and then press `Edit->Create Subnode`. Then you get a dialog that allows you to create the new subnode.

## 4.5 Deleting nodes

Nodes can also be deleted within the model browser. Currently this functionality is restricted to the deletion of transition nodes and EET nodes. To delete a node first select the desired node and then click on `Edit->Delete Node`. Be aware that all subnodes (i.e. the subtree of the selected node) will also be deleted instantaneously.

# 5 Integration

## 5.1 Integration in AUTOFOCUS

### 5.1.1 Exporting Projects from AUTOFOCUS

You can export an AUTOFOCUS project by simply pressing `Export Project -> Quest Format` from the `Projects` menu in AUTOFOCUS and choosing a file name. The exporter creates one file for model with the given file name and for each specified DTD module a special module file. Note that all generated files have to be in the same directory to read or import the repository.

### 5.1.2 Importing Projects in AUTOFOCUS

Importing projects to AUTOFOCUS is as easy as exporting them. Just click on `Import Project->Quest Format` in the `Projects` menu and enter the file name that contains the model. Pay attention that the corresponding DTD module files are located in the same directory as the model file.

### 5.1.3 Limitations

Due to limitations of the quest data structures and the AUTOFOCUS data structures some information gets lost when converting a project from Quest to AUTOFOCUS or vice versa.

The Quest export can not handle the following items:

- refinement of axes in EETs (i.e. sub EETs for axes)
- textual annotations of all objects
- reuse of STDs in several components
- STD documents which are not associated with an SSD document
- combinations of input/output statements that are delimited with “ , ” can not be exported (use “ ; ” as delimiter)

Items that cannot be imported in AUTOFOCUS:

- all kinds of test data
- properties
- abstractions
- java code

Therefore, when exporting an AUTOFOCUS projects to the Quest format and reimporting it again in AUTOFOCUS, you will automatically loose the refinement of axes in EETs.

## 5.2 Integration in VSE

One part of Quest is the connection to the VSE [Pla97] [KBR98] tool. In this section we give a short overview, how VSE is integrated in AUTOFOCUS/Quest. Then we describe how projects or part of projects are imported to VSE or exported from VSE. At last some limitations are shown.

### 5.2.1 Overview

The integration of VSE is loose, which leads to the fact, that the user has to follow some rules. But the user is guaranteed the greatest flexibility. Figure 5 shows the process from AUTOFOCUS/Quest to VSE II and back. As it can be seen, there are three programs involved. For the translation of AUTOFOCUS/Quest-specifications to VSE the Quest-Selector can be used to select some parts of a project, which should be translated. After this the appropriate generator can be started to generate a VSE-SL [UBR+99] file which contains all the information. This file can be imported to VSE. For the retranslation the VSE-specification has to be exported and the appropriate generator has to be started. The resulting Quest repository can be imported to AUTOFOCUS/Quest afterwards.

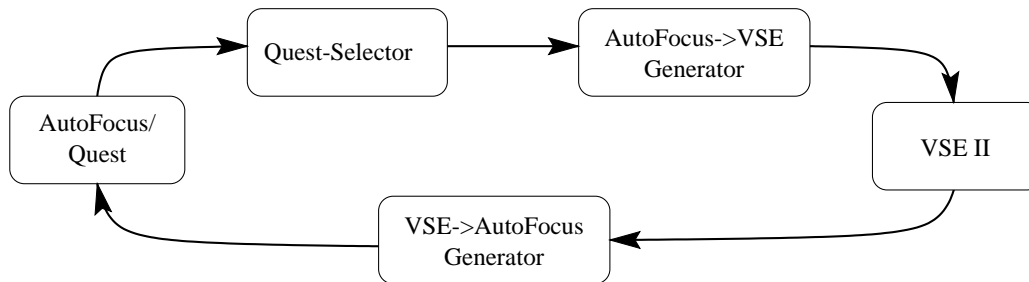


Figure 5: Connection AUTOFOCUS-VSE

## 5.2.2 Importing Projects in VSE

If you want to import an AUTOFOCUS-project to VSE, you have to export it first from AUTOFOCUS. For further help on exporting a project in the Quest-format see 5.1.1. Now the selector can be used to choose a part of the project, which should be translated to VSE or the translation can be started on the whole project.

**Translation of a whole project** To generate a VSE-SL file for the whole project start the following command in a shell<sup>6</sup>:

```
af2vse [-useinclude] [-withoutsubgraphs]
      <QML-filename> <VSE-filename> [<PropertyFilename>]
```

You have to specify the filename of the AUTOFOCUS/Quest repository and the filename of a VSE-SL file in which the translated code should be saved. Also a file with properties (in text-format) may be specified which then will be translated to VSE. There are two other optional arguments which control the structure of the generated VSE-SL code. The option `-useinclude` controls if the translation of an AUTOFOCUS component with its automaton will be inlined or included in the scheduled component wrapper (see Section 8 for further details).

**Translation of selected parts** To select parts of a specification, which should be exported to VSE II, the selector<sup>7</sup> has to be started.

```
af2vse -s[elect] [-useinclude] [-withoutsubgraphs]
      [<QML-filename> [<VSE-filename> [<PropertyFilename>]]]
```

<sup>6</sup>Note: The start of `af2vse` should be possible from the Quest-Browser in later versions.

<sup>7</sup>Note: This will be possible from the Quest-Browser in a later version.



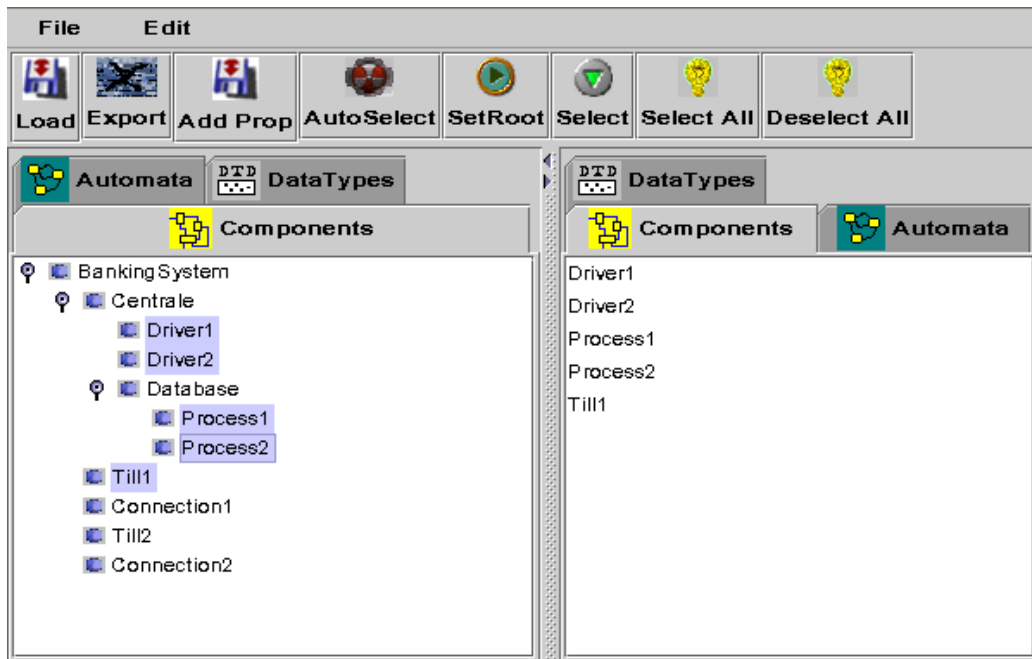


Figure 6: The Selector

Note, when using the selector it is not necessary to specify any filename on the command line. An AUTOFOCUS/Quest specification filename can be selected in a file requester later by clicking on the button `Load` or by selecting the `Load` in the `File` menu. If properties are stored in a separate file, they can load with `Add Prop`.

The selector is presented in Fig. 6. There is a menu bar and a toolbar, where some actions can be started. All actions, besides `Exit`, which is only present in the `File` menu, are available via the menus and via the toolbar. Below the toolbar there are two panels<sup>8</sup> with tabs in it. On the left side objects, like components, automata, data types and properties, can be selected on the right side the actual selection is shown. The right panel is there only to control the current selection, modifications always have to be done on the left side.

For maximal flexibility and usability there are two main selection modes. The automatic selection mode tries to select always a complete, consistent model, while the user has maximal freedom in the manual selection mode. First we describe the automatic selection mode.

<sup>8</sup>Panels are “windows”, which are contained in a real window

The automatic selection mode can be started by clicking on the toolbar button `AutoSelect` or by selecting the corresponding menu entry. If the automatic selection mode is on, the button `AutoSelect` is enlightened.

**Automatic selection mode:** The component tree will be folded to the root component. If the root component has an automaton, then it will be selected, otherwise their subcomponents will be unfolded. Again for each subcomponent, which has an automaton it will be selected. Each subcomponent without an behavior will be unfolded and so on. So at the beginning the most abstract system with a complete behavioral description and the necessary data type definitions will be selected. After this subtrees can be unfolded to get a more concrete selection. A left mouse button click, deletes the selection and starts the selection algorithm from the component, on which was clicked. A right mouse button click adds the selected component.

**Manual selection mode:** A left mouse button click deletes the selection and adds the component, on which the mouse was positioned. A right mouse button click selects or unselects the current object under the mouse. If a tree was folded all objects in it will be unselected.

With the `Select` button the current selection can be activated. So the selected objects in the left panel will be presented in lists in the right panel. These lists in the right panel can only be viewed to control the selection, no operation on the right side is possible.

With the `Select All` button the whole specification can be selected. The current selection can be deleted with the button `Deselect All`.

If you want to export only a subtree of the whole project you have to select one component (left click) and then you have to choose `Set Root`. Afterwards only the selected subtree will be presented. To reset this operation unselect all selected objects, e.g. with right mouse button clicks, and then click on `Set Root`.

To export the current selection presented in the right panel, choose `Export`. Afterwards a filename for the VSE-SL specification can be chosen and then the translation to VSE II starts.

**Import in VSE** After the appropriate VSE-SL file has been generated, it can be imported to VSE. To start the import use the menu `Import Quest` in the `File` menu of the VSE main window (see Figure 7). Then the following steps are performed:

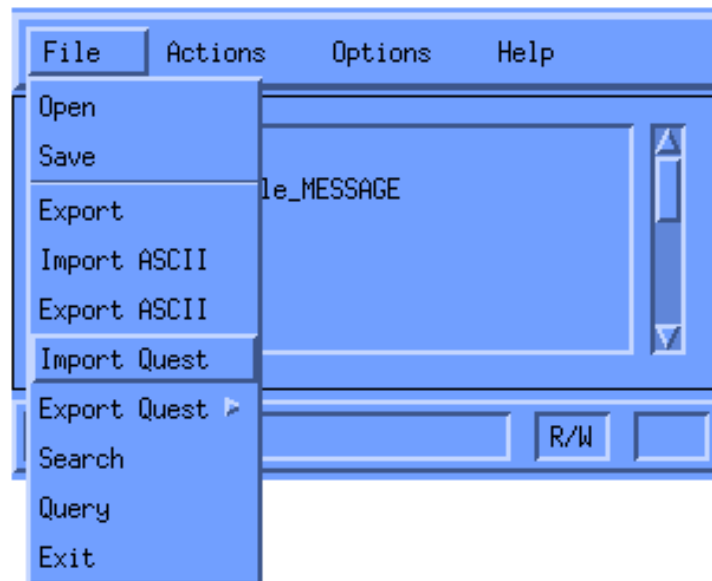


Figure 7: VSE: Import Quest

1. If the current VSE repository is no new one, i.e. it is already saved in a `.gt.i` file, then the user can choose to create a new empty project.
  - (a) If  **Yes** is selected, then a new project with the name " `noname` " will be created and used.
  - (b) If  **No** is selected, then the state of the current project will be saved to `/tmp/quest.bck` and all changes will be made to the current project.
2. The AUTOFOCUS/Quest repository can be chosen with the file-browser (default suffix is `.out`).
  - (a) Choose  **Ok** to import the file.
  - (b) Choose  **Cancel** to cancel the import operation.
3. The AUTOFOCUS/Quest repository will now be read. All actions, particularly error messages, will be stored in a log-file (`Quest.log` in the start directory).
4. If errors occurred during the import and the import is done in an existing VSE repository, then the user can choose to restore the previous version.

Some notes on VSE internals:

**Import of new nodes** If no node with the representation of a specification exists, then a new node with the appropriate name/class will be created. The initial node attributes are set like in a “normal” ASCII-import, in particular `Checked = No`.

**Change of existing nodes** If an existing specification will be imported, the status information of the node and dependent nodes will be recomputed from the original state and the kind of the change:

Original state	Action
Unchecked	the node will be changed without a request
Checked	the node will be changed without a request, the node and dependent nodes become Unchecked
Invalid	the node will be changed without a request, the node and dependent nodes become Unchecked
Suspended, Verified	the node will be type-checked automatically and if necessary a warning with the possibility to cancel the action will be shown. If the warning is ignored dependent nodes become Unchecked or Invalid.
In-Verification	the node won't be changed. This may lead to inconsistencies, because then parts of a AUTOFOCUS-specification are imported and other parts aren't.

### 5.2.3 Exporting Projects from VSE

If you want to export a project or parts of a project from VSE and reimport it to Quest you have to select the appropriate VSE objects, and you must choose `Export` from the `File` menu of the VSE main window. Note that the retranslation to AUTOFOCUS/Quest can be started with `Call Transformer` from the VSE main window (see 9) automatically. After that the resulting specification can be read from the Quest-tools or from AUTOFOCUS.

**Selection** Before the export to Quest can be started, the VSE development objects, which should be exported, have to be selected. To select some nodes you have to double click on a node and to choose a selection operation. Selected nodes are displayed inverted. Select operations add nodes to the already selected nodes.

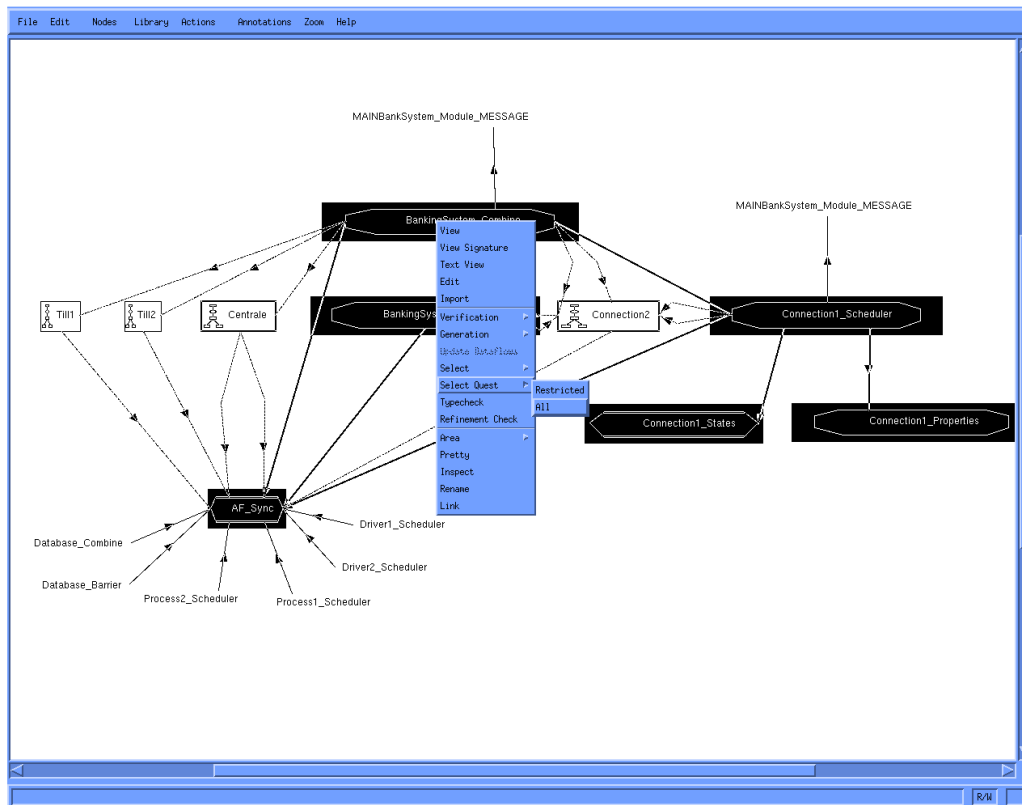


Figure 8: VSE: Select Quest

So more than one select operation may be used and the operations to describe the set of objects to be exported may be mixed. There are two special operations for the use with Quest. The popup menu of Theory and Tlspec contains a sub-menu `Select Quest` with the operations `Restricted` and `All` (see Figure 8). With `All` all objects are selected, with `Restricted` all objects in the graph which are referenced from the actual object (including the actual object itself) are selected. Since it is not always obvious, which nodes are selected you may want to choose `Deselect All` in `Actions` menu of the VSE main window, before you begin your selection.

**Export** After you have chosen all objects you want to reimport to Quest, the the export can be started with `File` only or `Call Transformer` in the `Export Quest` submenu of the VSE main window (see Figure 9). With `File` only you have to start the VSE to `AUTOFOCUS` translation yourself, if you

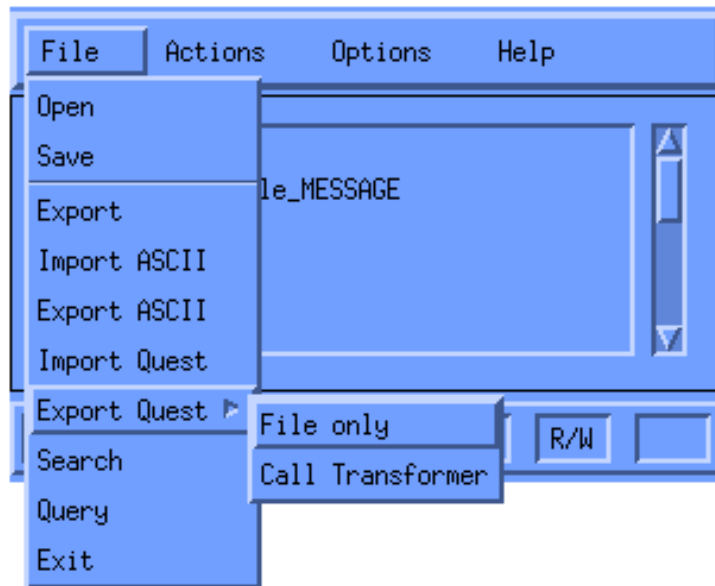


Figure 9: VSE: Export Quest

choose `Call Transformer` then the retranslation to Quest is started automatically. Before the export itself is started, some heuristic checks are done:

- Type correctness: every selected node is type checked
- Completeness: all nodes used by a selected node are selected too.
- Coherence: only one root node is selected. All other nodes are reachable from the root node.

If a check fails the user is warned, but may proceed anyway.

The start of the retranslation program can be configured via the environment variables `VSE2AF_BIN` and `VSE2AF_PAR`. The translation program will be started as:

```
$VSE2AF_BIN $VSE2AF_PAR /tmp/vse2af.out
```

See below and Section [A](#) for further information on the translation program and how the environment variables should be configured.

**Retranslation** If you have exported VSE objects with `File only` or you want to retranslate an existing VSE-SL file to Quest you can use the command:

```
vse2af <VSE-filename> <QML-filename>
```

The filename of a VSE-SL file and the filename to which the translation should be written have to be specified.

**Import** After these steps the resulting AUTOFOCUS repository file can be imported to AUTOFOCUS or to the Quest-tools.

#### 5.2.4 Limitations

The retranslation from VSE to Quest is limited to VSE-SL specifications, which are similar structured like those generated from AUTOFOCUS/Quest specifications. For more details see Section 8.

## 6 Model Checking with AUTOFOCUS

In this section we describe how to use model checking with the models of AUTOFOCUS. The description of properties is in Section 3.5. After a general introduction (Section 6.1), we describe how counter examples can be visualized (Section 6.2). In the following (Section 6.3.1, and Section 6.3.2) we describe how the the different checkers can be used. The technical details of the translations are described in other documents. See [PS99] for the SMV connection, and [Wim00] for the translation to SATO.

### 6.1 Introduction

In this section we give general information for model checking, such as parameter selections for model checking and bounded model checking.

For model checking only finite types are allowed. Since many models use integer values, we decided to provide an ad hoc abstraction for them. We implemented the checking of QuestF integers within a range from 0 to the value of `MaxInt`. This can be used to benchmark different model checkers by changing the value of `MaxInt`. This can be done within the “Verify” menu of the model browser. Since

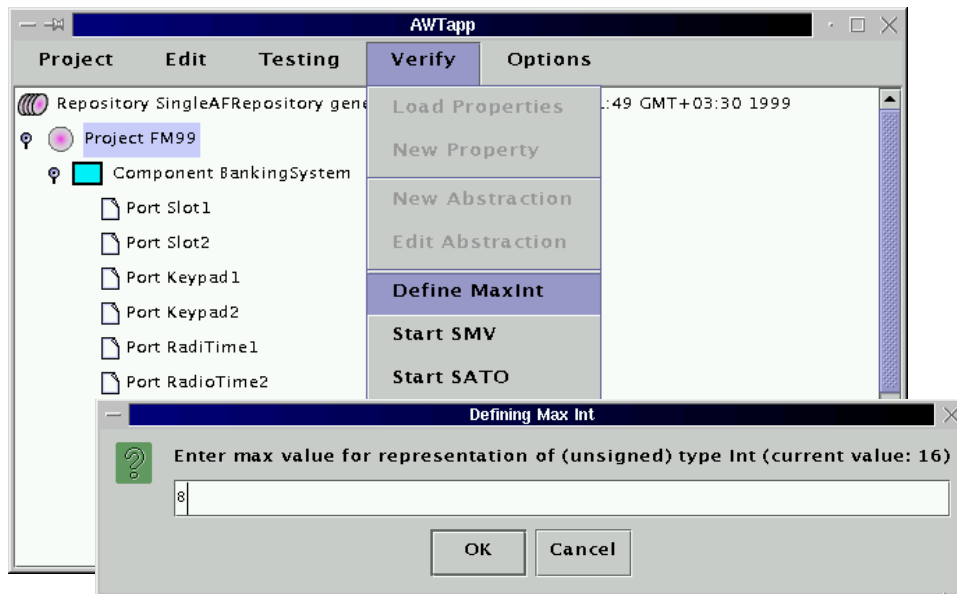


Figure 10: Changing MaxInt

MaxInt is an attribute of projects a project has to be selected. The default value for MaxInt is 15. See Figure 10 for a changing of MaxInt.

Model checking (and bounded model checking) can be started for components and for projects. Be careful with this selection, since it can influence the truth of properties. Checking a component includes all subcomponents, but not the surrounding components. Model checking evaluates all properties of the selected components. Model checking assumes an arbitrary environment, that sends arbitrary inputs to the checked component. Therefore there can be counter examples that are no real counter examples, i.e. that are not possible for the complete system. Just think of two components, one generating the constant output true, the other, an identity component, just passes boolean values. A property of the identity component is that it does not sent always true, however this is not the case in this example, where only true arrives from the other component.

This effect can be exploited to reduce the search space of a system, by providing an “environment model” for a system. This will lead to faster checking results, even if the number of components if bigger. Experiments however have shown, that a too detailed environment model increases the transition relation and thus can make checking more time consuming.

Since model checking (and bounded) model checking are of exponential complexity, it is possible to enter a maximum time bound for the checking time. If



this time bound is exceeded, checking is aborted. In addition it is possible to cancel the model checking manually. The selection of the time bound is done after the `start SMV` or the `start SATO` in the verify menu. The relation from the passed time and the maximum time is displayed in a progress bar. Of course it is possible to run several checks in parallel, however not of the same component.

For bounded model checking a search bound is required. This reduces the search to counter examples of this length. If the search bound is greater than the diameter of a system, than the result of bounded model checking is equal to model checking, i.e. if no counter examples are found than the checked properties are verified. The diameter is the maximum of the shortest path to every reachable state.

## 6.2 Counter Examples

Model checking and bounded model checking produce counter examples. They are inserted automatically into the model browser in form of a MSC. The name of the generated MSC corresponds to the property. The attributes like channels and messages can be inspected using the model browser.

To visualize the counter examples graphically (as EET views of the MSC models), the repository has to be saved and imported from the AUTOFOCUS tool.

## 6.3 Connected Checkers

Within the project Quest two checkers have been connected to AUTOFOCUS. SMV and SATO. In this section we describe some technical details of these connections, that are not relevant to the user of AUTOFOCUS/Quest, but they help to understand intermediate files that are generated (and not removed) for documentation. Furthermore it might be interesting for experts to see, and edit the generated files and to restart model checking.

### 6.3.1 Using SMV

After the determination of the maximal time, a progress control window appears, that contains information of the current steps. This window disappears after the maximal time, or if cancel is pressed. An example for this window is in Figure 11.

For experts: the SMV Translation is done in four automatic steps, with three intermediate files, that can be inspected:

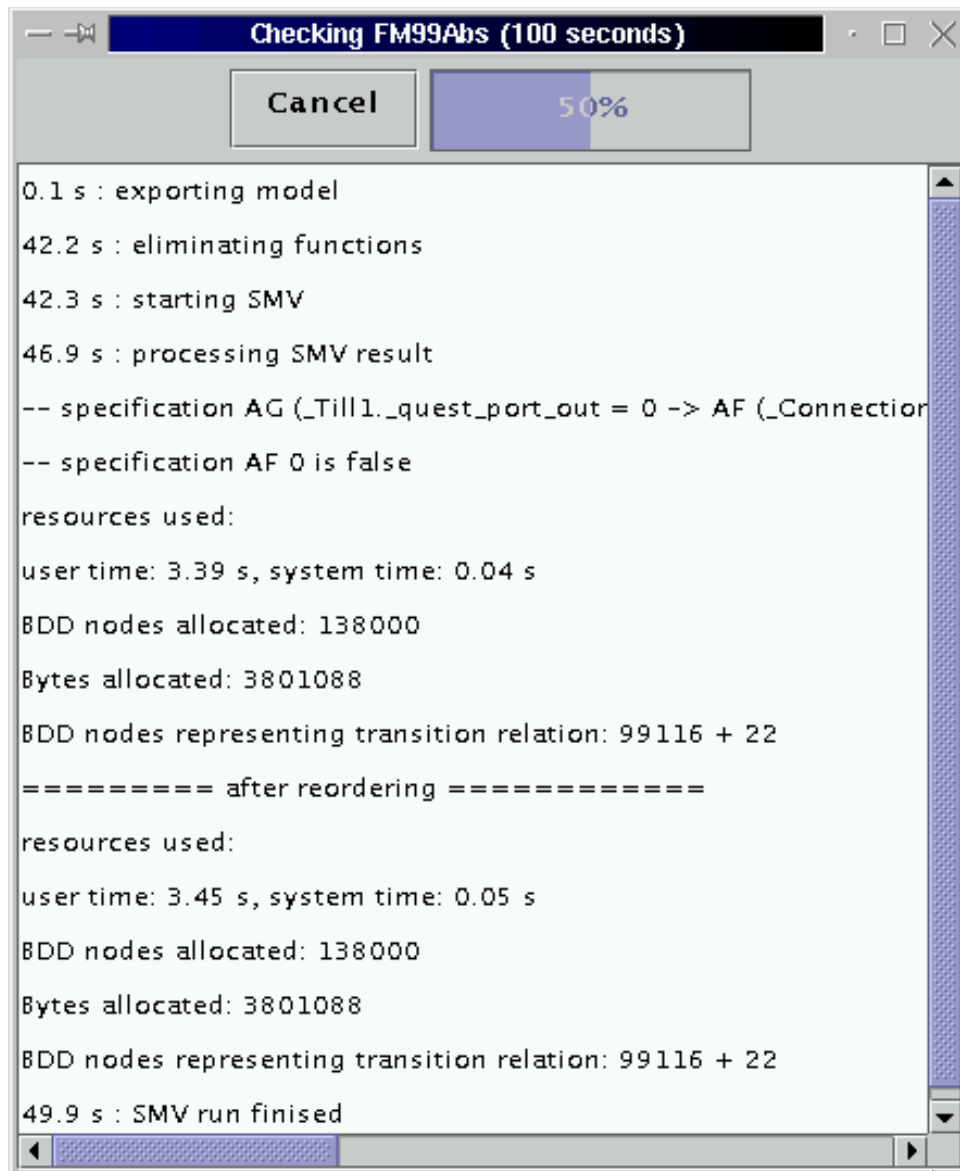


Figure 11: Model Checking Information

1. `name.names`: name mapping from AUTOFOCUS/Quest to SMV
2. `name.c`: the model with cpp macros (not always generated)
3. `name.smv`: the model with expanded macros
4. `name.counter`: the result of SMV (with counter examples)

where *name* is the name of the model concatenated with the name of the checked component.

### 6.3.2 Using SATO

SATO works completely automatic and does not generate intermediate files.

## 7 Abstraction Chooser

Model checking is completely automatic, but limited to systems of relatively small finite state space. Theorem proving, however, is in a sense complementary: it requires user interaction but can deal with systems of arbitrary size. Abstractions provide conditions that ensure the correctness of such simplifications, and so abstraction techniques [CGL92, LGS<sup>+</sup>95, GL93, Mül98b, Mer97, Kur87, DGG97, SLW95, Wol86, MN95, HS96] promise to combine the advantages of both approaches. Within the project Quest we decided to support abstractions related to certain properties, this gives us a very powerful tool for reducing complex systems, to their critical core.

First we will propose a customized abstraction methodology for the context of Quest. Then we describe tool support for defining abstractions (see Section 7.3). In Section 7.2, we present an example and the resulting proof obligations.

### 7.1 Methodology

The general idea of abstraction is depicted in Fig. 12: in a first step, the original system  $C$  is reduced to a smaller model  $A$ . If  $C$  is large or infinite, this step will in general require interactive proof techniques. In a second step, the smaller system  $A$  is analyzed using automatic tools.

Usually, the smaller system  $A$  is obtained by partitioning the state space of  $C$  via a function  $h$  between the two state spaces [CGL92]. If  $h$  is a homomorphism

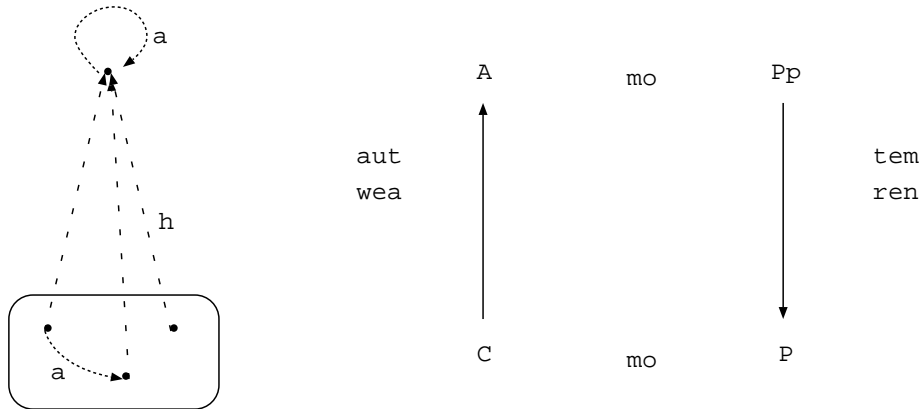


Figure 12: Abstraction

(*abstraction function*), the abstraction is guaranteed to be sound, i.e. if the abstract system satisfies a property so does the original system.

In the following we sketch how those abstraction techniques could be integrated into the process model and tool environment of Quest. We propose the following methodology, that allows to generate appropriate abstractions incrementally. It is an instance of the more general methodology for abstractions developed in [Mül98b].

- Start with a primitive abstraction function for a safety property.
- If model checking fails, the reason may be either that the desired property is refuted already or that the abstraction is not sophisticated enough. This question should be answered by testing/simulating the generated counterexample w.r.t. the original automaton.
  1. If we feel that the generated counterexample is indeed a counterexample in the concrete system, we have to check this in the concrete system. Here, we have to consider that the generated counterexample is given in terms of the abstract system, i.e. in order to run it on the concrete system we first have to replace the objects in the abstract system by one of its preimages in the concrete system, i.e. we have to invert the abstraction function. Since the inversion of the abstraction function may lead to many preimages the test might require to backtrack among the different alternatives until the error is located within the concrete model.
  2. If no error can be found when simulating the system, the error is (if all relevant cases have been considered) in the abstraction function. In

this case the abstraction should be improved. The simplest possibility is to come up with a completely new abstraction function that works in the desired way. However, there are several more refined possibilities for such an improvement. See [Mül98b] for details.

When repeating this step this results in an incremental process that finally reaches an appropriate abstraction.

- The correctness of this abstraction function then should be shown by theorem-proving. In the context of Quest, this should be done by the VSE II tool component. The criteria for the correctness of the abstraction functions *abs* are

1. if  $s^c$  is an initial state the  $abs(s^c)$  is also an initial state, in  $abs(s^c)$ .
2. if  $t^c$  is a successor of  $s^c$  the  $abs(t^c)$  is a successor of  $abs(s^c)$ .

See [Mül98b] for the mathematical definitions of the correctness, and Appendix C.1 for an example.

This methodology enables the reuse of simple abstraction functions in order to come up with a powerful and sufficient abstraction concept and thus decreases the amount of intuition usually required for abstractions.

The abstraction function relates concrete and abstract components in AUTOFOCUS/Quest. This requires that the abstraction function relates

- control states,
- local variables,
- input and output ports,
- types with the following informations:
  - name of the type,
  - constants and functions, and
  - a mapping from concrete to abstract values
- the relevant properties.

Therefore abstraction functions consist of several parts, that all have to be defined, and that have to fit together, for example the mapping of the ports has to respect the mapping of the types.

According to our methodology, abstraction functions are an attribute of the concrete component, therefore abstraction functions are treated as attributes of the concrete component in the model browser.

To facilitate the input of abstraction functions Quest provides an “abstraction chooser” that helps in defining type correct abstractions. First (in Section 7.2) we shortly present a small example for abstractions, and in Section 7.3 we explain how to use the abstraction chooser.

## 7.2 Abstraction Example: Comparator

The comparator has been modeled as a part of the storm surge barrier case study. This comparator compares the inner water level with the outer water level, to determine whether the barrier can be opened. This is the case if the outer water level plus a fixed difference is lower than the inner water level. Water levels are measured by sensors sending integer values to the comparators. For the critical properties of the system, it suffices to differentiate only a few number of integer ranges. Therefore we defined the following type for the abstract system model.

```
data SensorSig = None | SNeg | S00 | S25 | S30;
```

The structure of both systems is in Figure 13. The type Signal has only one

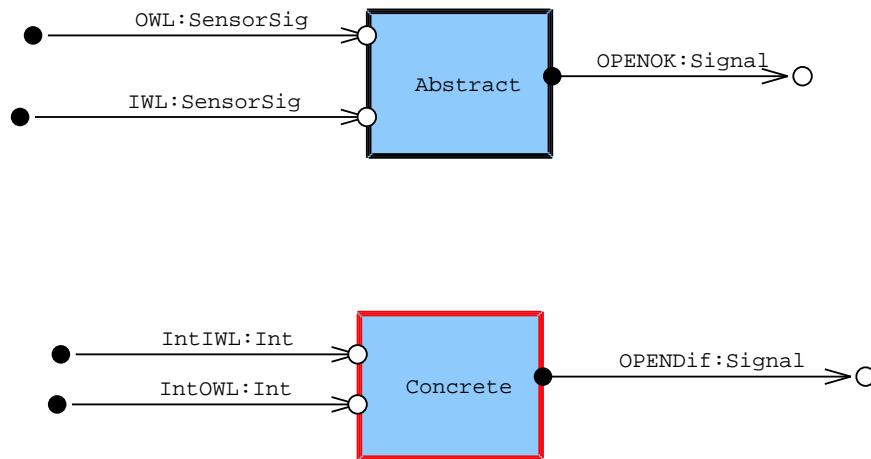


Figure 13: Interfaces of Abstract and Concrete Models

signal element (Present) to indicate that the comparison is positive.

The behaviour of the system is quite simple (see Figure 14 for the concrete and Figure 15 for the abstract model). It consists of an STD with only one state and one transition, however within the transition there are constants (openDifference), and operations (+, <) that have to be abstracted as well. Therefore we defined the following operations on the type SensorSig:

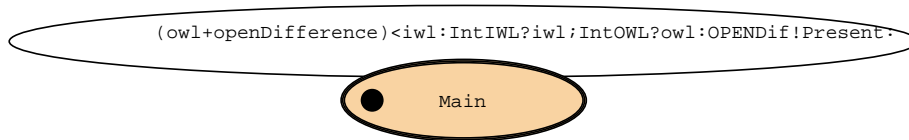


Figure 14: Concrete Behaviour

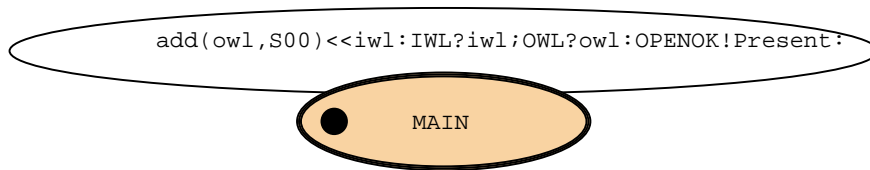


Figure 15: Abstract Behaviour

```

op << : SensorSig -> SensorSig-> Bool;
fun None << x = False
  | x << None = False
  | x << SNeg = False
  | S00 << S00 = False
  | S25 << S00 = False
  | S25 << S25 = False
  | S30 << x = False
  | x << y = True;

add : SensorSig -> SensorSig -> SensorSig;
fun add(x,S00) = x
  | add(S00,x) = x
  | add(x,y) =None;

```

On the concrete system we want to check the property

```
[ ]((Val(IntOWL) < Val(IntIWL)) => <>(OPENDif!Present))
```

In terms of the abstract model this property is

```
[ ]((Val(OWL) << Val(IWL)) => <>(OPENOK!Present))
```

To ensure that the concrete property holds, we have to show that the abstract property implies the concrete one, i.e. that is stronger. The homomorphism property of the abstraction function (the relation between the concrete and abstract states and transitions) states that every concrete step (including the idle steps) has a corresponding abstract step. We generate one VSE II theory for the definition of the abstraction (see Appendix C.1), one theory for the homomorphism predicate, and one for the satisfies relation (see Appendix C.2). Our experience with similar proofs in the Isabelle system [Ham98, Mü198b] showed, that the correctness conditions for the homomorphism predicate can be proved easily mainly using simplification. With the abstraction chooser of Section 7.3 it is very easy to define correct abstractions.

The homomorphism property allows us to eliminate the temporal operators, and to concentrate on the correctness of the pure formulas for proving strengthenings.

To show the strengthening, of an implication, we have to strengthen the correctness conditions for this implication is a strengthening of the right conclusion `OPENOK!Present => OPENDif!Present` and a weakening of the conditions `Val(OWL) << Val(IWL) <= Val(IntOWL) < Val(IntIWL)`.

For the definition of strengthenings in VSE II (see Appendix C.3) the definition of the elements mapping from concrete to abstract elements is required. The correctness conditions are in a separate theory, as well as the satisfies statement (see Appendix C.3).

Our experiences showed, that proving strengthenings mostly involves simple reasoning on data structures, for which the VSE II system is very appropriate. However, it takes much time to realize that an abstraction is not correct, especially on the level of theorem provers. Therefore the most important step is to define correct abstractions. This is done on the modelling level and not within the theorem prover.

### 7.3 Using the Abstraction Chooser

To define an abstraction in Quest, the concrete component has to be selected. If an abstraction already exists it is stored as an attribute of this component, and can



be edited. In this section we show how to define a new abstraction. After the concrete component is marked, in the “Verify” menu the button “New Abstraction” is enabled, that starts the abstraction chooser.

In general there are two possibilities to define abstractions:

1. to compute the abstract component from an abstraction function, and
2. to define the abstraction function between a concrete and an abstract model.

In Quest (up to now) only the second way is supported, because designers mostly have the abstract model (in mind) and want to find a homomorphism between the concrete and the abstract one. Defining such an abstraction function in general is more difficult (due to the consistency conditions), and therefore Quest supports difficult task with the abstraction chooser. The feature of computing abstraction functions is future work.

To select the abstract component for the abstraction the abstraction chooser shows all components in the system (see Figure 16). This requires that both components (the concrete and the abstract) are specified within one SSD.

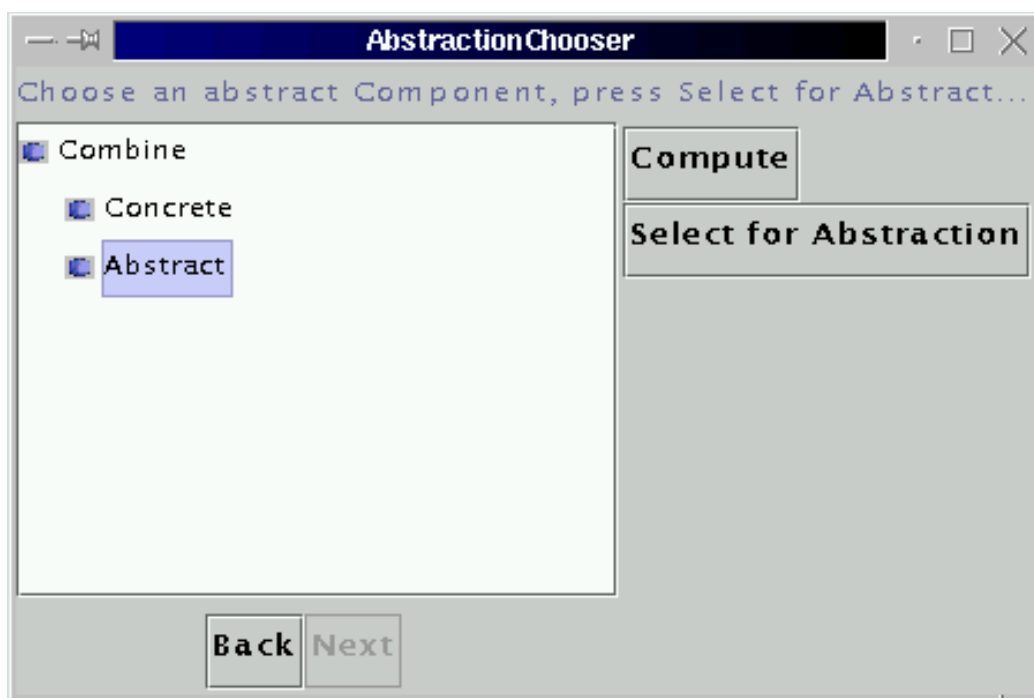


Figure 16: Selecting the Abstract Component

After the selection of the abstract component all elements (Variables, Ports, Types, ..) that can be related are defined, and the definition window of the abstraction chooser appears (see Figure 17). It suggests an order for the definition of the abstraction functions, but allows for arbitrary orders as well. Furthermore suggestions are made that consider the previous definitions. In Figure 17 the abstraction

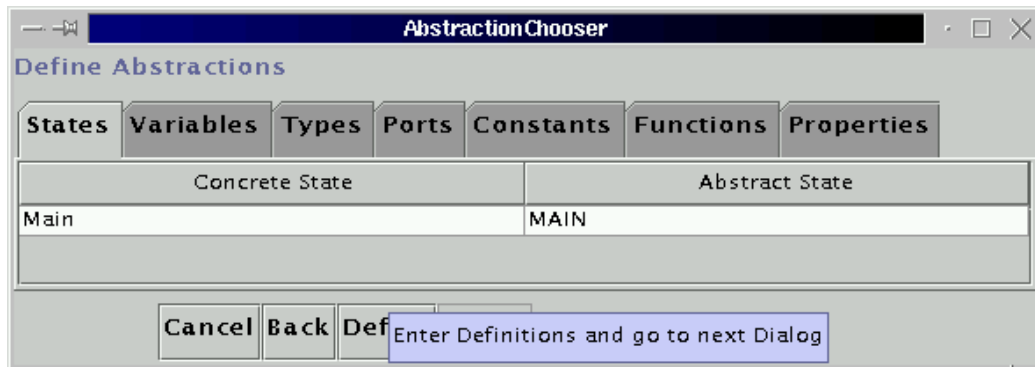


Figure 17: Definition Window of Abstraction Chooser

chooser suggests to map the concrete state `Main` to the abstract state `MAIN`. In this example this is the only possible choice, and pressing “Define” defines the abstraction function for states. After the definition of the mapping between local variables (in the example there is also no choice), the chooser tries to make a suggestion for the abstraction of types that matches to the abstracted types of the ports. (see Figure 18). If there are inconsistencies, for example if the type of a

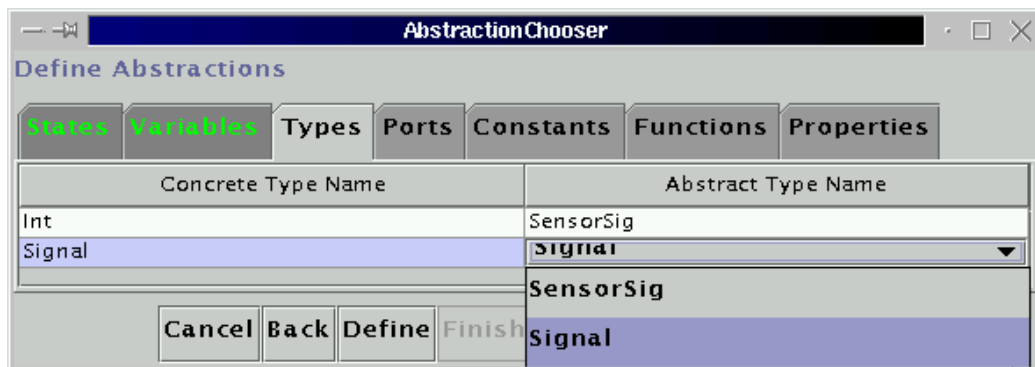


Figure 18: Selecting the Abstract Types

function abstraction does not match the type define for local variables, the error

is reported, and the inconsistent fields are marked with red colors in the definition window (see Figure 19).

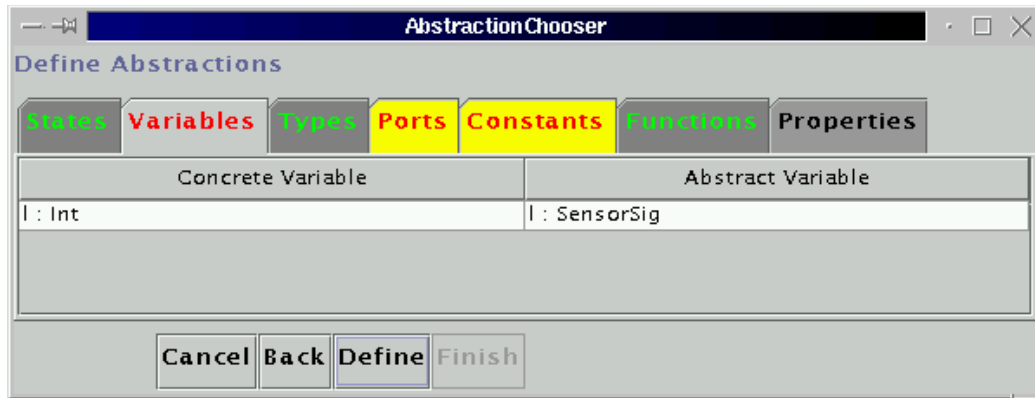


Figure 19: Inconsistencies are marked with red color

The functions that map the values of the concrete type to the abstract (in our case `Int2SensorSig`) are defined in the “Functions” window of the chooser, that allows to enter a text that defines the abstraction function. Pattern matching for abstraction functions is not supported in this version.

An important part of the definition of the abstraction functions is the relation of properties. The correctness of the abstraction function is defined only for the related properties. For them strengthening proof obligations are generated. In principle, every concrete property could be abstracted to arbitrary abstract properties, however a similar temporal structure is recommended, i.e. an  $[ ] ( \dots \Rightarrow \langle \rangle ( \dots ) )$  property should be related to a property with a similar structure, otherwise the strengthening proof obligation can become very difficult.

The abstraction chooser displays for each property of the concrete component a choose box with all properties entered to the abstract component (and the value **not related**). The abstraction chooser allows to enter a related property for each concrete, in our example only one property is related (see Figure 20). If all parts of the abstraction function are defined correctly, the “Finish” button is enabled. Pressing it stores the abstraction to the concrete component. The proof obligations for the correctness are generated, when the repository is exported to VSE II.

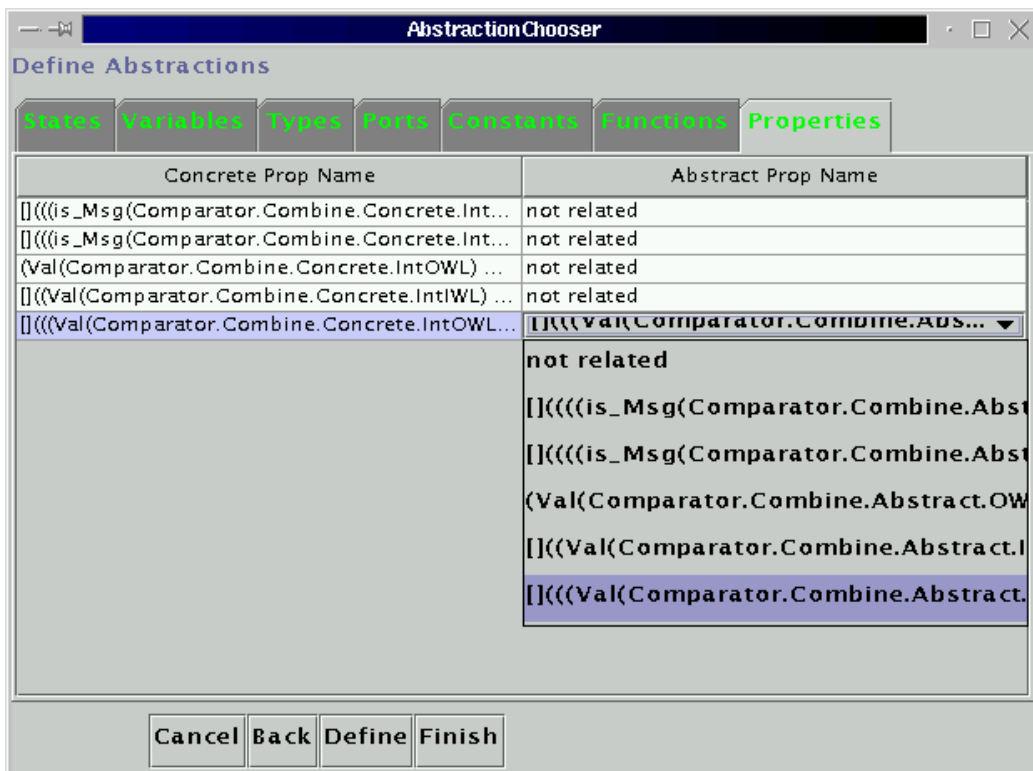


Figure 20: Relating the Properties

## 8 Connection to VSE

In this section we describe the translation from AUTOFOCUS to VSE and the retranslation. We show how the different views of AUTOFOCUS are translated and how the retranslation works in detail.

### 8.1 Introduction

In AUTOFOCUS there are four different views on systems. System structure diagrams (SSDs) describe the static aspects of a distributed system by a network of connected components, exchanging data over channels. State transition diagrams (STDs), which are similar to finite automata, are used to describe the dynamic aspects, i.e. the behavior, of the components. Extended event traces (EETs) are used to describe exemplary runs of a system from a component based view. Datatype definitions (DTDs) define the types of data processed by a system and are used in the other views. The used views are hierarchical. For a further description of AUTOFOCUS and its views see [HS97] and [HMS+98] and [HEea96].

In AUTOFOCUS/Quest we use these views to model systems, which are based on a synchronous execution model where time is divided into a sequence of intervals. In each interval, a channel in the system can carry at most one value, and each component executes a reaction, where input values are read from the component's input ports, and output values are written on the component's output ports. Data values on ports that are not read by a component are lost. Output written by a component in one reaction is visible to other components only in the next interval. If for a component no reaction is explicitly specified, it remains in its current state and produces no output. This default behavior is referred to as an idle transition.

TLA, which is used in the VSE II system, is based on an asynchronous interleaving semantics. This mismatch between synchronous and interleaving semantics can be solved by augmenting the system with a scheduling algorithm. In order to keep the translated specifications as modular as possible we use a hierarchic scheduling algorithm. The order in which parallel components executes their reaction is arbitrary. So a well known scheduling algorithm, the barrier synchronization algorithm is used.

The connection to VSE uses only SSDs, STDs and DTDs. The datatype defined in DTDs are translated directly to VSE and are retranslated in a similar way from VSE. The (hierarchical) structure of a system will be presented in VSE completely. Behavior of an AUTOFOCUS component will be present in the appropriate VSE component.

Each scheduled component in a group of concurrently executing scheduled components perform their reaction in a sequence of phases. It is required that no scheduled component begin executing its  $(p + 1)^{th}$  phase until all scheduled components in its group have completed their  $p^{th}$  phase. Each scheduled component has a variable `schedulePhase`, which stores the phase which the scheduled component would enter. For the synchronization only two different phases are needed. Internally each interval is split in two phases, the compute phase, where the input is read and the output is computed, and the copy phase, where the computed output is published to the other components. Each scheduled component of a system must either finish its compute phase or its copy phase before a component can enter the next phase. The activation of this two phases is done by the barrier. The barrier has an internal variable `phase`, which stores the current phase of the system and a counter, which stores the number of components which have not finished the current phase. The barrier has two methods `BeginSync` and `EndSync`. The general scheme of the barrier synchronization algorithm is shown below<sup>9</sup>:

```

PROCESS Component
  DATA
    schedulePhase : boolean INITIAL true
  SPEC
    WHILE true DO
      Barrier.BeginSync(schedulePhase)
      IF schedulePhase = true THEN
        Compute
      ELSE
        Copy
      FI
      Barrier.EndSync()
      schedulePhase := ~schedulePhase
    OD
END

PROCESS Barrier
  DATA
    phase : boolean INITIAL true
    compCounter : nat INITIAL N /* number of
                                synchronized components */
  ACTIONS

```

---

<sup>9</sup>We use a pseudo language to shorten the description

```

BeginSync(schedulePhase : boolean) ::=
  phase = schedulePhase
EndSync ::=
  IF compCounter = 1 THEN
    phase := ~phase
    AND compCounter := N
  ELSE
    compCounter := compCounter - 1
  FI
END

```

## 8.2 Translation from AUTOFOCUS to VSE

In this section the way how AUTOFOCUS/Quest specifications are translated to VSE is shown. As mentioned above we translate SSDs, STDs and DTDs to VSE<sup>10</sup>.

### 8.2.1 SSD

At first we show the translation of SSDs. SSDs are somehow the main part of the translation, because the components contain or include all other translated parts. As mentioned above in AUTOFOCUS we use a synchronous execution model, while VSE is based on an asynchronous interleaving semantics. So we have to use the barrier synchronization scheduling algorithm, which was described above.

In the following examples we show how this algorithm is used concretely.

The first example shows a simple SSD with two components `Connection1` and `Central` (see Fig. 21. This is a clipping of the example presented in Section B.2. In this example we translated the SSD as if both components had an associated automaton. The automaton is not shown in the code example presented below.

First the BASIC type `AF_Sync`<sup>11</sup> is defined. `tsync` is used for the communication between the barrier and the schedulers.

After this the global synchronization barrier `BankingSystem0_Barrier`<sup>12</sup> is defined. As in its scope there are two components, its internal variable `compCounter` is initialized with 2. The global phase is initialized with T, which

<sup>10</sup>Properties entered in the Quest-Browser are also translated to VSE

<sup>11</sup>Note: Prefixes `AF_` and `AFi_` are used to mark Quest internal objects

<sup>12</sup>Note, that sometimes numbers are appended to a name, to avoid name clashes.

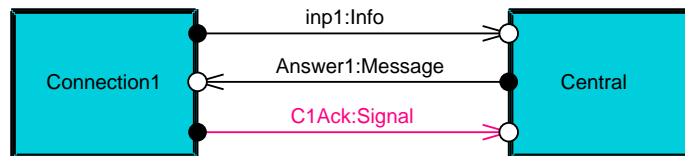


Figure 21: Banking System with two components

means that the compute phase is active. For the communication with the schedulers the shared inout variables `sync_Connection1` and `sync_Central` and the in variables `phase_Connection1` and `phase_Central` are used. The `sync` variables are set to `BSync` and `ESync` from the schedulers and are reset to `Ok` by the barrier. With `BSync` and `ESync` a scheduler signals, that it wants to begin a new synchronization phase or that it wants to end its current phase. In the barrier this leads to the execution of the appropriate action `BeginSync` and `EndSync`. Note that the `BeginSync` action is guarded by `phase = ph`, which means that the global phase must be the same as the phase which the scheduled component wants to enter. The `Sync` and the `SyncAll` actions are only abbreviations used to make the code more compact.

```

BASIC AF_Sync
  tsync = BSync |
          ESync |
          InitSync |
          Ok
BASICEND

/*
 * AutoFocus Barrier for BankingSystem
 */

TLSPEC BankingSystem0_Barrier

  PURPOSE "AutoFocus Barrier for

```



```

Component BankingSystem0"

USING Boolean; Natural; AF_Sync

DATA
  SHARED INOUT sync_Connection1 : tsync
  IN phase_Connection1 : bool

  SHARED INOUT sync_Central : tsync
  IN phase_Central : bool

  INTERNAL compCounter : nat
  INTERNAL phase : bool

ACTIONS
  BeginSync(ph : IN bool) ::=
    phase = ph
    AND UNCHANGED(compCounter, phase)

  EndSync ::=
    IF compCounter = 1 THEN
      phase' = ~phase
      AND compCounter' = 2
    ELSE
      compCounter' = compCounter - 1
      AND UNCHANGED(phase)
    FI

  Sync(s : OUT tsync, ph : IN bool
    , s1 : OUT tsync) ::=
    (s = BSync AND BeginSync(ph) AND s' = Ok
    OR s = ESync AND EndSync AND s' = Ok)
    AND UNCHANGED(s1)

  SyncAll ::= Sync(sync_Connection1,
    phase_Connection1,
    sync_Central)
    OR Sync(sync_Central,
    phase_Central,
    sync_Connection1)

```

```

SPEC
  INITIAL
    compCounter = 2
    AND phase = T /* ComputePhase */
    AND sync_Connection1 = InitSync
    AND sync_Central = InitSync

  TRANSITIONS [ SyncAll ]
    {compCounter, phase, sync_Connection1,
     sync_Central}
  FAIRNESS WF(SyncAll)
    {compCounter, phase, sync_Connection1,
     sync_Central}

  HIDE compCounter, phase

TLSPECEND

```

Next the scheduler for the component `Connection1` and the component itself is shown. In this example there are two separate TLSPECs but these two could also be merged to one component (see Section 5.2.2 and the notes on `-useinclude`). In the scheduler `Connection1_Scheduler` it can be seen, that the description of the component `Connection1` is included. The scheduler has an out variable `schedulePhase`, which presents the value of the current phase, i.e. T (compute) or F (copy). Also a shared inout variable `sync` is used for the communication with the global barrier. The scheduler has three actions, `WaitOk`, `Copy` and `Compute`. The `WaitOk` action waits for an `Ok` signal on `sync`, which can only be set from the barrier. The `Copy` and the `Compute` action are executed in the appropriate phase. Note, that in the example below the `Compute` action is “empty”, because for simplicity reasons no automaton for `Connection1` is included.

Initially the `schedulePhase` is set to T as the global phase. `sync` is set to `InitSync`, so that there is a defined value. The scheduler then enters an endless loop, where it tries to enter the next phase by setting `sync` to `BSync`. It waits for a `Ok` from the barrier and then executes the appropriate action. After this it signals the end of his phase to the barrier.

In TLSPEC `Connection1` the ports `Answer`, `inp`, and `CAck` of its component are defined. The type of these ports is derived from the appropriate channel type (see Section 8.2.3). For all out ports an internal port is defined, which is used as a buffer of length one. The `NextStep` action and the coding of the automaton of `Connection1` is missing in the example below (see Section 8.2.2). This

action uses the internal ports for storing the computed results. The CopyPorts action copies these results to the external ports, which can be seen from other components.

```
/*
 * AutoFocus Scheduler for Component: Connection1
 */

TLSPEC Connection1_Scheduler

PURPOSE "AutoFocus Scheduler for
        Component Connection1"

USING
  /* ImportDataTheories */

  Natural
  ; Boolean
  ; AF_Sync
  ; MAINCHANNEL

INCLUDE
  AF_Component = Connection1

DATA
  OUT schedulePhase : bool
  SHARED INOUT sync : tsync

ACTIONS
  WaitOk ::= sync = ok
           AND UNCHANGED(schedulePhase, sync)

  /* CopyPorts */
  Copy ::= AF_Component.CopyPorts
           AND schedulePhase' = T
           AND UNCHANGED(sync)

  /* Compute */
  Compute ::= schedulePhase' = F AND UNCHANGED(sync)

SPEC
  INITIAL BEGIN
```

```

        schedulePhase := T; /*Compute*/
        sync := InitSync
    END

    TRANSITIONS BEGIN
        WHILE (TRUE) DO
            sync := BSync;
            WaitOk;
            IF schedulePhase = T THEN
                Compute
            ELSE
                Copy
            FI;
            sync := ESync;
            WaitOk
        OD

    END {schedulePhase, sync,
        AF_Component.AFi_ControlState,
        inp, AFi_inp, CAck, AFi_CAck}
    TLSPECEND

/*
 * AutoFocus Component: Connection1
 */

    TLSPEC Connection1

    PURPOSE "AutoFocus Component Connection1"

    USING
        /* ImportDataTheories */

        Natural
        ; MAINCHANNEL

    DATA
        /* PortDeclarations */

        IN Answer : Channel_Message.ChannelBase.Channelm
        OUT inp1 : Channel_Info.ChannelBase.Channelm
        OUT CAck : Channel_Signal.ChannelBase.Channelm

```

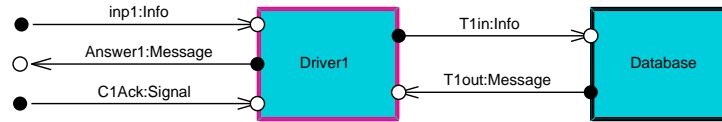


Figure 22: Central

```

/* PortDeclarations (internal-ports) */

INTERNAL AFi_inp :
    Channel_Info.ChannelBase.Channelm
INTERNAL AFi_CAck :
    Channel_Signal.ChannelBase.Channelm

ACTIONS
    CopyPorts ::=
        inp1' = AFi_inp1 AND CAck' = AFi_CAck
        AND UNCHANGED(AFi_inp1, AFi_CAck)

/* Automaton definition is missing
   for simplicity reasons */

TLSPECEND

```

The component Central is decomposed in two components Driver1 and Database (see Fig. 22). So the component Central is a “combination” of those two components, as the root component BankingSystem is a combination of Connection1 and Central\_Combine. In components, which are decomposed, the ports are inlined in the TLSPEC of the Combine component. In the Combine components all “external” variables are connected. Beside the in and out variables for component ports, like `inp1` and `inp`<sup>13</sup>, the communication channels between the schedulers and the barriers are connected.

```

/*
 * Combine for AutoFocus Component: Central

```

<sup>13</sup>The port `inp1` of the component Central is connected to the port `inp` of the subcomponent Driver1 through the channel `inp1`. The names of the ports are not shown in Fig. 22

```

*/

TLSPEC Central_Combine

PURPOSE "AutoFocus Combine for Component Central"

USING
  /* ImportDataTheories */

  Natural
  ; AF_Sync
  ; MAINCHANNEL

DATA
  INTERNAL sync_Driver1 : tsync
  INTERNAL sync_Database : tsync
  SHARED INOUT sync : tsync
  OUT schedulePhase : bool

  /* PortDeclarations */

  IN inp1 : Channel_Info.ChannelBase.Channelm
  IN ClAck : Channel_Signal.ChannelBase.Channelm
  OUT Answer1 : Channel_Message.ChannelBase.Channelm

COMBINE
  Central_Barrier [
    Central_Barrier.phase
      -> Central_Combine.schedulePhase,
    Central_Barrier.phase_Driver1
      <- Driver1_Scheduler.schedulePhase,
    Central_Barrier.phase_Database
      <- Database_Scheduler.schedulePhase
  ] SHARED [
    Central_Barrier.upsync
      <- Central_Combine.sync,
    Central_Barrier.sync_Driver1
      <- Central_Combine.sync_Driver1,
    Central_Barrier.sync_Database
      <- Central_Combine.sync_Database
  ]

```

```

;Driver1_Scheduler [
    Driver1_Scheduler.AF_Component.inp
        <- Central_Combine.inp1,
    Driver1_Scheduler.AF_Component.CAck
        <- Central_Combine.ClAck,
    Driver1_Scheduler.AF_Component.Answer
        -> Central_Combine.Answer1,
    Driver1_Scheduler.AF_Component.Tout
        <- Database_Scheduler.AF_Component.Tlout ]
SHARED [
    Driver1_Scheduler.sync
        <- Central_Combine.sync_Driver1 ]

;Database_Scheduler [
    Database_Scheduler.AF_Component.Tlin
        <- Driver1_Scheduler.AF_Component.Tin ]
SHARED [
    Database_Scheduler.sync
        <- Central_Combine.sync_Database ]

HIDE Central_Combine.sync_Driver1,
      Central_Combine.sync_Database

TLSPECEND

/*
 * Combine for AutoFocus Component: BankingSystem
 */

TLSPEC BankingSystem0_Combine

PURPOSE "AutoFocus Combine for
        Component BankingSystem0"

USING
    /* ImportDataTheories */

Natural
; AF_Sync
; MAINCHANNEL

```

```

DATA
  INTERNAL sync_Connection1 : tsync
  INTERNAL sync_Central : tsync

COMBINE
  BankingSystem0_Barrier [
    BankingSystem0_Barrier.phase_Connection1
      <- Connection1_Scheduler.schedulePhase,
    BankingSystem0_Barrier.phase_Central
      <- Central_Combine.schedulePhase
  ] SHARED [
    BankingSystem0_Barrier.sync_Connection1
      <- BankingSystem0_Combine.sync_Connection1,
    BankingSystem0_Barrier.sync_Central
      <- BankingSystem0_Combine.sync_Central
  ]

;Connection1_Scheduler [
  Connection1_Scheduler.AF_Component.Answer
    <- Central_Combine.Answer1 ]
  SHARED [
    Connection1_Scheduler.sync
      <- BankingSystem0_Combine.sync_Connection1 ]

;Central_Combine [
  Central_Combine.inp1
    <- Connection1_Scheduler.AF_Component.inp,
  Central_Combine.C1Ack
    <- Connection1_Scheduler.AF_Component.CAck ]
  SHARED [
    Central_Combine.sync
      <- BankingSystem0_Combine.sync_Central ]

HIDE BankingSystem0_Combine.sync_Connection1,
      BankingSystem0_Combine.sync_Central

TLSPECEND

```

As we use a hierarchic barrier synchronization algorithm, the component `Central` has its own barrier. This barrier communicates with the global barrier like



a normal scheduler. It acts to the local schedulers like the global barrier. But internally it differs from the global barrier. It has an additional internal variable `endcomp` which is T when all of its scheduled components have performed their step and the barrier itself signals the end of its step to his super-barrier. In contrast to the root barrier, which only performs `SyncAll` actions, sub-barriers have to synchronize with their super-barriers. So they have a `TRANSITIONS` section like schedulers.

```

/*
 * AutoFocus Barrier for Central
 */

TLSPEC Central_Barrier

  PURPOSE "AutoFocus Barrier for Component Central"

  USING Boolean; Natural; AF_Sync

  DATA

    SHARED INOUT sync_Driver1 : tsync
    IN phase_Driver1 : bool

    SHARED INOUT sync_Database : tsync
    IN phase_Database : bool

    INTERNAL compCounter : nat
    INTERNAL endcomp : bool
    OUT phase : bool
    SHARED INOUT upsync : tsync

  ACTIONS

    BeginSync(ph : IN bool) ::=
      phase = ph
      AND UNCHANGED(compCounter, phase, endcomp, upsync)

    EndSync ::=
      IF compCounter = 1 THEN
        phase' = ~phase
        AND compCounter' = 2
        AND endcomp' = T
        AND UNCHANGED(upsync)
      ELSE

```

```

        compCounter' = compCounter - 1
        AND UNCHANGED(phase, endcomp, upsync)
    FI

WaitOk ::= upsync = ok AND
    endcomp' = F AND
    UNCHANGED(compCounter, phase, upsync,
        sync_Driver1, sync_Database)

Sync(s : OUT tsync, ph : IN bool
    , s1 : OUT tsync) ::=
    (s = BSync AND BeginSync(ph) AND s' = Ok
    OR s = ESync AND EndSync AND s' = Ok)
    AND UNCHANGED(s1)

SyncAll ::= Sync(sync_Driver1, phase_Driver1,
    sync_Database)
    OR Sync(sync_Database, phase_Database,
    sync_Driver1)

SPEC
INITIAL BEGIN
    compCounter := 2;
    endcomp := F;
    phase := T; /* ComputePhase */
    sync_Driver1 := InitSync;
    sync_Database := InitSync
END

TRANSITIONS BEGIN
    WHILE TRUE DO
        upsync := BSync;
        WaitOk;
        WHILE (endcomp = F) DO
            SyncAll
        OD;
        upsync := ESync;
        WaitOk
    OD
END {compCounter, phase, upsync, endcomp,
    sync_Driver1, sync_Database}

```

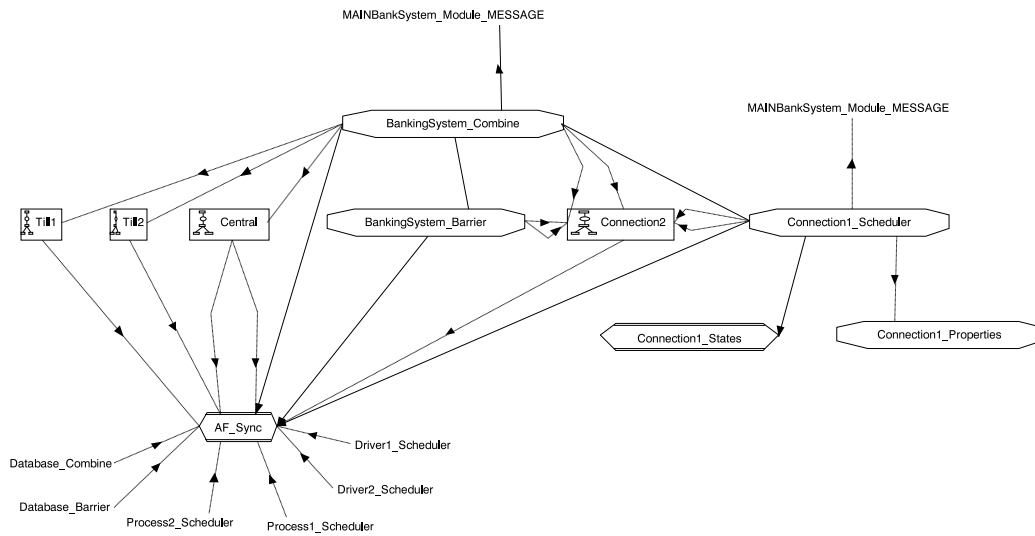


Figure 23: Development Graph of the Banking System

```
HIDE compCounter, endcomp
```

```
TLSPECEND
```

The missing TLSPECs for the components `Driver1`, and `Database` of the example are similar to the TLSPECs of the component `Connection1` and are therefore not presented here.

In Fig. 23 the development graph of the whole example like presented in Appendix B.2 is shown.

### 8.2.2 STD

In this section we explain the translation of components with automaton to VSE-SL. As in the example above we use the `withincludes` option, so that the scheduler and the automaton of a component are split into two separate TLSPECs. This is simpler for understanding and editing the generated specification, but it may be more unhandy when doing proofs.

In the following example the component `Driver1` is shown. In Fig. 22 the SSD of the component `Central` with the component `Driver1` is shown. In Fig. 24 the automaton, which describes the behavior of `Driver1` is presented. Below the VSE-SL specification of the scheduler and the component itself is shown. The

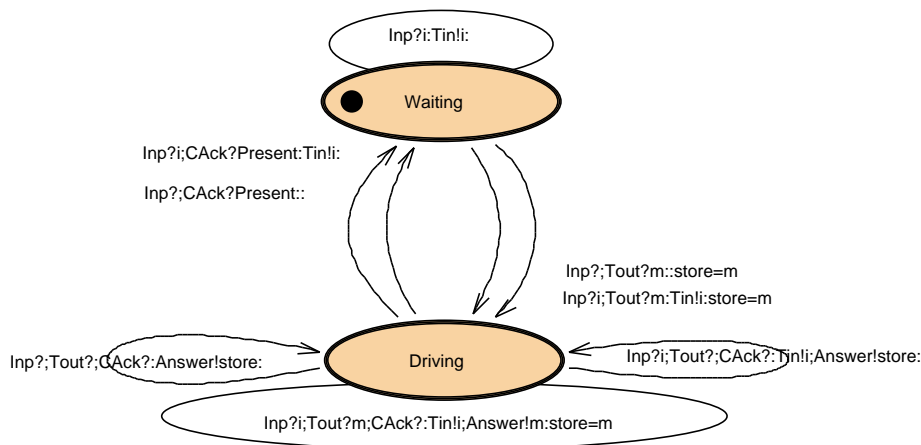


Figure 24: Automaton of Component Driver1

scheduler is similar to the scheduler of component `Connection1`. The component `Driver1` is included in the scheduler as `AF_Component`. The `Compute` action calls the `NextStep` action of the component. This action performs one step of the automaton or, if no transition is possible, then the `IdleStep` is taken. The `Copy` action calls the `CopyPorts` action of the component, which copies the values of the internal buffers to the out variables.

In `Driver1_States` the states of the automaton `Waiting` and `Driving` are defined. These are used as values for the `AFi_ControlState`, which stores the current state of the automaton. As described above, besides the in and out variables of the component, internal buffer variables for the out variables are defined in the component (for a description of the used channel type see Section 8.2.3). The result of a compute action is stored in their internal buffers, so that they cannot be seen outside the component. In the `CopyPorts` action these internal values are copied to the external out variables. For every transition of the automaton an action, which describes the transition is generated. The action is named by the start state and the end state of a transition. For hierarchical automata the transition segments are merged, so that there is only one transition and so that the resulting automaton is flat. In a transition first the control state is checked

and set appropriate, and after this the precondition, the input-pattern, the outputs, and the actions are added. An extra transition, the `IdleStep` is added after the other transitions. If for a compute phase no transition could be taken, then the automaton stays in its current state, the local variables are unchanged and the out variables are set to `NoVal`. So the condition for an `IdleStep` is the negotiation of all preconditions and all input patterns of all transitions. The `NextStep` action calls one of the transitions or the `IdleStep` and so performs one reaction of the automaton. Initially the control state, and the local variables, and the out variables of the component are set appropriate.

```

/*
 * AutoFocus Scheduler for Component: Driver1
 */

TLSPEC Driver1_Scheduler

    PURPOSE "AutoFocus Scheduler for Component Driver1"

    USING
        Natural
        ; Boolean
        ; MAINBankSystem_Module_MESSAGE
        ; AF_Sync

    INCLUDE
        AF_Component = Driver1

    DATA
        OUT schedulePhase : bool
        SHARED INOUT sync : tsync

    ACTIONS
        WaitOk ::= sync = ok AND UNCHANGED(schedulePhase, sync)

        /* CopyPorts */
        Copy ::= AF_Component.CopyPorts
                AND schedulePhase' = T AND UNCHANGED(sync)

        /* Compute */
        Compute ::= AF_Component.NextStep
                AND schedulePhase' = F AND UNCHANGED(sync)

    SPEC

```

```

INITIAL BEGIN
    schedulePhase := T; /*Compute*/
    sync := InitSync
END

TRANSITIONS BEGIN
    WHILE (TRUE) DO
        sync := BSync;
        WaitOk;
        IF schedulePhase = T THEN
            Compute
        ELSE
            Copy
        FI;
        sync := ESync;
        WaitOk
    OD

    END {schedulePhase, sync,
        AF_Component.AFi_ControlState,
        store, Tin, AFi_Tin, Answer, AFi_Answer}
HIDE AF_Component.AFi_ControlState, store,
    AF_Component.AFi_Tin, AF_Component.AFi_Answer

TLSPECEND

BASIC Driver1_States
    tDriver1_States = sWaiting | sDriving
BASICEND

/*
 * AutoFocus Component: Driver1
 */

TLSPEC Driver1

    PURPOSE "AutoFocus Component Driver1"

    USING
        Natural
        ; MAINBankSystem_Module_MESSAGE
        ; Driver1_States

```

## DATA

```
/* PortDeclarations */
```

```
IN Tout : Channel_Message.ChannelBase.Channelm  
IN Inp  : Channel_Info.ChannelBase.Channelm  
IN CAck : Channel_Signal.ChannelBase.Channelm  
OUT Tin  : Channel_Info.ChannelBase.Channelm  
OUT Answer : Channel_Message.ChannelBase.Channelm
```

```
/* PortDeclarations (internal-ports) */
```

```
INTERNAL AFi_Tin  : Channel_Info.ChannelBase.Channelm  
INTERNAL AFi_Answer : Channel_Message.ChannelBase.Channelm
```

```
/* LocVariables */
```

```
INTERNAL store : Message
```

```
/* ControlState */
```

```
INTERNAL AFi_ControlState : tDriver1_States
```

## ACTIONS

```
CopyPorts ::=
```

```
  Tin' = AFi_Tin AND Answer' = AFi_Answer  
AND UNCHANGED(AFi_ControlState, AFi_Tin, AFi_Answer, store)
```

```
/* Transitions */
```

```
Waiting2Driving ::= AFi_ControlState = sWaiting  
  AND AFi_ControlState' = sDriving  
  AND Channel_Info.ChannelBase.is_Msg( Inp )  
  AND Channel_Message.ChannelBase.is_Msg( Tout )  
  AND ( AFi_Tin' = Channel_Info.ChannelBase.Msg(  
    Channel_Info.ChannelBase.Val( Inp )) )  
  AND AFi_Answer' = Channel_Message.ChannelBase.NoVal  
  AND ( store' = Channel_Message.ChannelBase.Val( Tout ) )
```

```
Waiting2Waiting ::= AFi_ControlState = sWaiting
```

```

AND AFi_ControlState' = sWaiting
AND Channel_Info.ChannelBase.is_Msg( Inp )
AND ( AFi_Tin' = Channel_Info.ChannelBase.Msg(
      Channel_Info.ChannelBase.Val( Inp )) )
AND AFi_Answer' = Channel_Message.ChannelBase.NoVal
AND UNCHANGED( store )

Waiting2Driving_1 ::= AFi_ControlState = sWaiting
AND AFi_ControlState' = sDriving
AND Channel_Info.ChannelBase.is_NoVal( Inp )
AND Channel_Message.ChannelBase.is_Msg( Tout )
AND AFi_Tin' = Channel_Info.ChannelBase.NoVal
AND AFi_Answer' = Channel_Message.ChannelBase.NoVal
AND ( store' = Channel_Message.ChannelBase.Val( Tout ) )

Driving2Driving ::= AFi_ControlState = sDriving
AND AFi_ControlState' = sDriving
AND Channel_Info.ChannelBase.is_Msg( Inp )
AND Channel_Message.ChannelBase.is_Msg( Tout )
AND Channel_Signal.ChannelBase.is_NoVal( CAck )
AND ( AFi_Tin' = Channel_Info.ChannelBase.Msg(
      Channel_Info.ChannelBase.Val( Inp )) )
AND ( AFi_Answer' = Channel_Message.ChannelBase.Msg(
      Channel_Message.ChannelBase.Val( Tout )) )
AND ( store' = Channel_Message.ChannelBase.Val( Tout ) )

Driving2Driving_1 ::= AFi_ControlState = sDriving
AND AFi_ControlState' = sDriving
AND Channel_Info.ChannelBase.is_Msg( Inp )
AND Channel_Message.ChannelBase.is_NoVal( Tout )
AND Channel_Signal.ChannelBase.is_NoVal( CAck )
AND ( AFi_Tin' = Channel_Info.ChannelBase.Msg(
      Channel_Info.ChannelBase.Val( Inp )) )
AND ( AFi_Answer' = Channel_Message.ChannelBase.Msg(store) )
AND UNCHANGED( store )

Driving2Driving_2 ::= AFi_ControlState = sDriving
AND AFi_ControlState' = sDriving
AND Channel_Info.ChannelBase.is_NoVal( Inp )
AND Channel_Message.ChannelBase.is_NoVal( Tout )
AND Channel_Signal.ChannelBase.is_NoVal( CAck )
AND ( AFi_Answer' = Channel_Message.ChannelBase.Msg(store) )
AND AFi_Tin' = Channel_Info.ChannelBase.NoVal

```



```

AND UNCHANGED( store )

Driving2Waiting ::= AFi_ControlState = sDriving
AND AFi_ControlState' = sWaiting
AND Channel_Info.ChannelBase.is_Msg( Inp )
AND Channel_Signal.ChannelBase.is_Msg( CAck )
AND is_Present( Channel_Signal.ChannelBase.Val( CAck ) )
AND ( AFi_Tin' = Channel_Info.ChannelBase.Msg(
      Channel_Info.ChannelBase.Val( Inp ) ) )
AND AFi_Answer' = Channel_Message.ChannelBase.NoVal
AND UNCHANGED( store )

Driving2Waiting_1 ::= AFi_ControlState = sDriving
AND AFi_ControlState' = sWaiting
AND Channel_Info.ChannelBase.is_NoVal( Inp )
AND Channel_Signal.ChannelBase.is_Msg( CAck )
AND is_Present( Channel_Signal.ChannelBase.Val( CAck ) )
AND AFi_Tin' = Channel_Info.ChannelBase.NoVal
AND AFi_Answer' = Channel_Message.ChannelBase.NoVal
AND UNCHANGED( store )

IdleStep ::= UNCHANGED(AFi_ControlState, Tin, Answer)
AND NOT(Channel_Info.ChannelBase.is_Msg( Inp )
AND Channel_Message.ChannelBase.is_Msg( Tout )
AND NOT(Channel_Info.ChannelBase.is_Msg( Inp )
AND NOT(Channel_Info.ChannelBase.is_NoVal( Inp )
AND Channel_Message.ChannelBase.is_Msg( Tout )
AND NOT(Channel_Info.ChannelBase.is_Msg( Inp )
AND Channel_Message.ChannelBase.is_Msg( Tout )
AND Channel_Signal.ChannelBase.is_NoVal( CAck ))
AND NOT(Channel_Info.ChannelBase.is_Msg( Inp )
AND Channel_Message.ChannelBase.is_NoVal( Tout )
AND Channel_Signal.ChannelBase.is_NoVal( CAck ))
AND NOT(Channel_Info.ChannelBase.is_NoVal( Inp )
AND Channel_Message.ChannelBase.is_NoVal( Tout )
AND Channel_Signal.ChannelBase.is_NoVal( CAck ))
AND NOT(Channel_Info.ChannelBase.is_Msg( Inp )
AND Channel_Signal.ChannelBase.is_Msg( CAck )
AND is_Present( Channel_Signal.ChannelBase.Val( CAck ) ))
AND NOT(Channel_Info.ChannelBase.is_NoVal( Inp )
AND Channel_Signal.ChannelBase.is_Msg( CAck )
AND is_Present( Channel_Signal.ChannelBase.Val( CAck ) ))

```

```

/* set output-ports to NoVal */

AND AFi_Tin' = Channel_Info.ChannelBase.NoVal
AND AFi_Answer' = Channel_Message.ChannelBase.NoVal

/* local variables stay unchanged */

AND UNCHANGED( store )

NextStep ::= (Waiting2Driving OR Waiting2Waiting
OR Waiting2Driving_1 OR Driving2Driving
OR Driving2Driving_1 OR Driving2Driving_2
OR Driving2Waiting OR Driving2Waiting_1 OR IdleStep)
AND UNCHANGED(Tin, Answer)

SPEC
AFi_ControlState = sWaiting
AND store = NoMoney
AND Tin = Channel_Info.ChannelBase.NoVal
AND AFi_Tin = Channel_Info.ChannelBase.NoVal
AND Answer = Channel_Message.ChannelBase.NoVal
AND AFi_Answer = Channel_Message.ChannelBase.NoVal

TLSPRECEND

```

### 8.2.3 DTD

In this section we describe the translation of data types and functions from the Quest language (see Section 3) to the VSE II specification language. Furthermore we describe the translation of properties to VSE.

Since data types and functions are executable within Quest, we translated using the “executable parts” of VSE II. Some concepts, like pattern matching or parameterization are different between Quest and VSE, and therefore they cannot be translated directly. In this section we explain the principle translation scheme, and the translation the special constructs. We explain the translation algorithms by means of examples.

The translation of identifiers is as directly as possible, i.e. if the translated function is no VSE keyword, it is used for the translation. For VSE keywords similar names are generated, however these generated identifiers are retranslated directly. For a

round trip engineering the VSE keywords should therefore be avoided.

**Translation of Simple Data Types** A simple data type in Quest, has no type arguments (like `List(a)`). An example for a simple data type is:

```
data Nat = Zero | Succ(pred:Nat);
```

Simple data types are translated into BASIC specifications. Since data types in Quest have canonical discriminators `is_Zero`, `is_Succ`, these are also generated. The resulting VSE specification is:

```
BASIC NatBase
  Nat =
      Zero WITH is_Zero
    | Succ(pred:Nat) WITH is_Succ
BASICEND
```

For every data definition one basic specification is generated.

**Translation of Structured Data Types** Structured data types are DTDs that import other DTDs (see Figure 4), or data type definition that use data type definitions of the same DTD. For example the DTD TRANSACTION of Appendix B.2 contains:

```
data Action = SendMail | Withdraw(Int) | ViewBallance;
data Info = TA(Action,account:Int);
```

The translation of this structured specification to VSE II is:

```
BASIC ActionBase
  USING
    INTEGER
  Action =
    SendMail WITH is_SendMail
  | Withdraw(WithdrawSell:Int) WITH is_Withdraw
  | ViewBallance WITH is_ViewBallance
BASICEND
```

```
BASIC InfoBase
  USING
```

```

    ActionBase ;
    INTEGER
Info =
    TA(TASell:Action, account:Int) WITH is_TA
BASICEND

```

This results into a more detailed structure of data types in VSE II compared with the structures of DTDs in Quest.

**Translation of Function Definitions** Functions can be defined in Quest using pattern matching. Pattern matching is expanded using explicit case distinctions with the generated discriminator and selector functions. This allows us to generate (executable) algorithms for functions using the FUNCDEF keyword. Since functions cannot be defined in BASIC theories, every DTD is translated into a THEORY that includes the BASIC theories for the data types. Functions with result type Bool are translated into predicates. An example from the banking system (see Section B.2) is:

```

module TRANSACTION =
data Action = SendMail | Withdraw(Int) | ViewBallance;
data Info = TA(Action,account:Int);
fun isMoneyRequest(TA(Withdraw(x),acc))=True
    | isMoneyRequest(x)=False;
end

```

It is translated into

```

THEORY TRANSACTION
    USING
        BOOLEAN;
        INTEGER;
        ActionBase ;
        InfoBase ;
    PREDICATES
        isMoneyRequest : Info
    VARS
        VlisMoneyRequest : Info
    AXIOMS
    FOR isMoneyRequest :
        DEFPRED isMoneyRequest(VlisMoneyRequest) <->

```

```

    IF is_TA( VlisMoneyRequest )
      AND is_Withdraw( TASell( VlisMoneyRequest ) )
    THEN TRUE
    ELSE FALSE
    FI
THEORYEND

```

The name of this theory is the name of the DTD.

**Translation of Polymorphic Data Types** Polymorphic data types are types that have arguments (type variables) like `List(a)`, `Set(a)`. Within Quest the translation of polymorphic data types is supported, even if the concept of parameterization in VSE II is slightly different. The most important example of polymorphic data types is

```
data Channel(m)= NoVal | Msg(Val:m);
```

It is in the VSE II translation of every AUTOFOCUS/Quest component, since the channels, and ports of AUTOFOCUS can be empty (represented by `NoVal`). For polymorphic data types the are parameter theory generated, that contain all parameters of the defined and imported data types. Therefore different parameters have to be used if they should not be unified. The translation of the type `Channel(m)` are the following VSE II specifications:

```

THEORY mParam
  TYPES          m
THEORYEND

```

```

BASIC ChannelBase
  PARAMS mParam
  Channelm =
    NoVal WITH is_NoVal
  |      Msg(Val:m) WITH is_Msg
BASICEND

```

```

THEORY CHANNEL
  PARAMS mParam
  USING
    BOOLEAN;
    INTEGER;

```

```

    ChannelBase [ m ]
THEORYEND

```

The type channel is instantiated for every defined data type in the main module of the translation, that is imported from the translations of the AUTOFOCUS/Quest components.

```

THEORY MAINFM99_Module_MESSAGE
  USING
    MESSAGE;
    TRANSACTION;
    SIGNAL;
    Channel_Bool = CHANNEL [ Bool ];
    Channel_Int = CHANNEL [ Int ];
    Channel_Message = CHANNEL [ Message ];
    Channel_Msg = CHANNEL [ Msg ];
    Channel_Action = CHANNEL [ Action ];
    Channel_Info = CHANNEL [ Info ];
    Channel_Keys = CHANNEL [ Keys ];
    Channel_Cardinfo = CHANNEL [ Cardinfo ];
    Channel_Card = CHANNEL [ Card ];
    Channel_Date = CHANNEL [ Date ];
    Channel_Signal = CHANNEL [ Signal ]
THEORYEND

```

**Translation of Properties** Properties in AUTOFOCUS/Quest hold at the specified states. No intermediate states are possible due to the synchronous semantics of AUTOFOCUS. Due to the differences in the interaction semantics (synchronous, asynchronous) schedulers are generated for the representation of AUTOFOCUS models in VSE II (see Section 8.1). In VSE II there are intermediate states that are not reachable in AUTOFOCUS. Therefore AUTOFOCUS/Quest properties hold only in these states and the properties have to be expanded. This allows also for a translation of the next state relation.

The example of Section 3.5 the following property has been given:

```

[] (FM99.BankingSystem.Connection1.Answer?NoMoney =>
    () (FM99.BankingSystem.Connection1.CentralMsg!NoMoney))

```

It is translated into the following VSE II specification:

```

TLSPEC Connection1_Properties
INCLUDE          PropertyBase = Connection1_Scheduler
SPEC
  [] PropertyBase.schedulePhase = T
  AND PropertyBase.schedulePhase' = F
  AND ( ( Channel_Message.ChannelBase.is_Msg(
          PropertyBase.AF_Component.Answer )
        AND Channel_Message.ChannelBase.Val(
          PropertyBase.AF_Component.Answer )
        = NoMoney )
    -> ( PropertyBase.AF_Component.CentralMsg'
        = Channel_Message.ChannelBase.Msg(NoMoney) )
  )
TLSPECEND

```

Again the patterns are eliminated, as described for function definitions.

### 8.3 Translation from VSE to AUTOFOCUS

In this section the retranslation of VSE specifications to AUTOFOCUS/Quest is discussed. The retranslation of specifications is only suggestive for specification or parts of specifications which are generated by Quest. This is due the fact, that AUTOFOCUS/Quest is based on a synchronous execution model, while VSE II uses an asynchronous interleaving semantic. Up to now there is no useful presentation of such specifications in AUTOFOCUS/Quest. So it is only possible to retranslate specifications which are structured like shown in Section 8.2.

So usually a specification of a system is started in AUTOFOCUS/Quest and translated to VSE II later, if some properties have to be shown. In VSE II the specification could be modified, but if it or parts of it should be retranslated the structure of the specification should follow the structure of the generated constructs. Usually only the behavior of a component will be changed.

#### 8.3.1 SSD

For the retranslation of SSDs only the following parts are recognized:

Location	Specification part
Combine, Component, Scheduler	components and SSDs
Component	local variables, i.e. internal variables, except those beginning with AFi_
Component	initial values of local variables
Component, Combine	in and out variables, which are retranslated to in and out ports
Combine	channels

The structure of the SSDs will be the same as those used by the translation. All view informations, e.g. positions of the drawn rectangles and so on, are lost.

It is not checked, if the scheduling algorithms fit to those used by the translation, i.e. the “internal” specification of the barriers and the schedulers are not checked. Changes there will be ignored without warnings.

### 8.3.2 STD

For the retranslation of STDs only the following parts are recognized:

Location	Specification part
Component, States	States
Component	Transitions, with preconditions, input patterns, output actions, and actions

All retranslated automatons are flat, even if the original automaton was an hierarchic one. All view informations, e.g. positions of the drawn ellipsoids, are lost.

### 8.3.3 DTD

The retranslation of data types is restricted to the subset of the generated constructs, and requires that the theories have the structure as described in the generation in Section 8.2.3. Furthermore pattern matching definitions cannot be inferred from the case distinctions. Compared which the large possibilities VSE II offers to define data types this is quite restricted, however the retranslation can be used to integrate changes into the Quest data definitions, or as a starting point for further developments that can be used for round trip engineering.

Due to the semantical differences (synchronous vs. asynchronous) between AUTO-FOCUS/Quest and VSE II property translation generates very complex terms that correspond to the properties. Up to now there is no method in VSE II to deal



with such “synchronous properties”. Therefore no retranslation of properties is supported.

## 9 The Test Environment

### 9.1 Introduction

The test environment of Quest enables you to perform the following tasks of the testing process:

- Derivation of test sequences from specifications
- Manual creation of test sequences
- Running the tests on test objects

Although the test environment has a test driver that allows you to perform the specified test sequences, the test environment focuses more on the creation of reasonable test sequences than on the execution of given tests. Fig. 25 shows an overview over all testing tasks of the Quest development process. The rectangles of the diagram are components of the Quest repository and the arrows are testing tasks that produce new components or alter existing components in the repository.

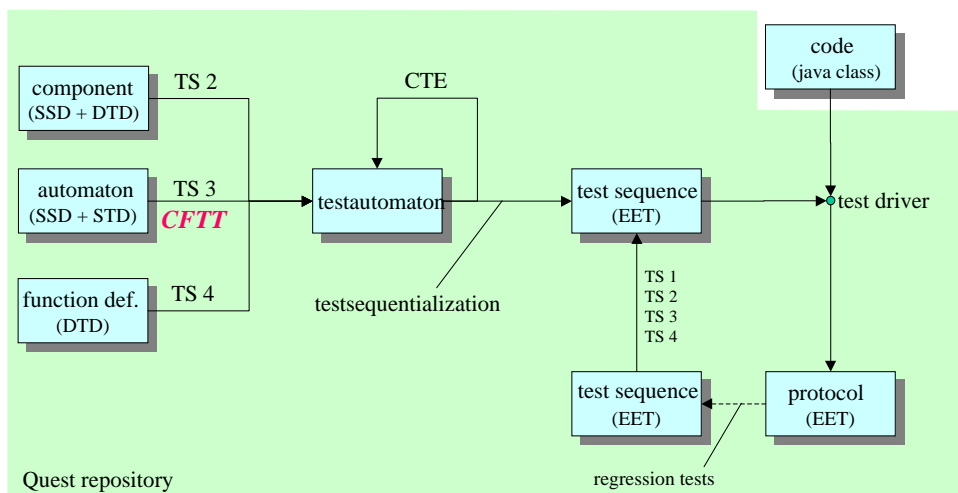


Figure 25: Overview about the test activities in Quest

## 9.2 Creating Test Sequences

There are several possibilities to create reasonable test sequences that can be used for testing the target software (the test object). The choice of the appropriate test sequence creation method is mainly determined by the type of the given specifications, i.e. the more detailed the specifications are, the more specific is the test sequence generation process. The kind of the given specification is called the test basis. In Quest, we distinguish the following test bases:

- component interfaces (i.e. specifications with SSDs and DTDs)
- components with behavior (i.e. SSDs, STDs and DTDs)
- function definitions (DTDs)
- sequence diagrams (EETs)

In the following sections we discuss the specific test creation methods depending on the given test basis. The central component is the test automaton. The test automaton is a temporary automaton that is used in the testing process. It represents the behavior or at least a part of the behavior of the test object. During the test case generation process the tester works with the test automaton. Quest's testing environments allows you to adapt the test automaton in a systematic way to resolve the needs for the software testing process. This task is supported by the classification tree editor (CTE) from DaimlerChrysler. Finally the tester can derive test sequences from the test automaton which will be displayed as sequence diagrams. Depending on the test automaton, the resulting sequence diagrams include the expected responses or not. If they do not include results, either a reference implementation is needed to produce the results or the tester can complement them manually.

### 9.2.1 Test Basis: SSDs

The System Structure Diagrams in particular include the interfaces of the components described by the specification. These interfaces are a good starting point for test case generation. The test case generation from component interfaces works as follows:

- STEP 1: produce CTA file<sup>14</sup> from component interface and data type definitions

---

<sup>14</sup>The cta file contains the information for the classification tree assistant in the cte

- STEP 2: start CTE with generated CTA-File and save test cases in CTE-File
- STEP 3: read CTE file and generate test automton for component

All three steps can be done in one guided procedure of the browser. First, mark the desired component in the browser window. Then click on `startCTE` in the testing-menu (see Fig. 26).

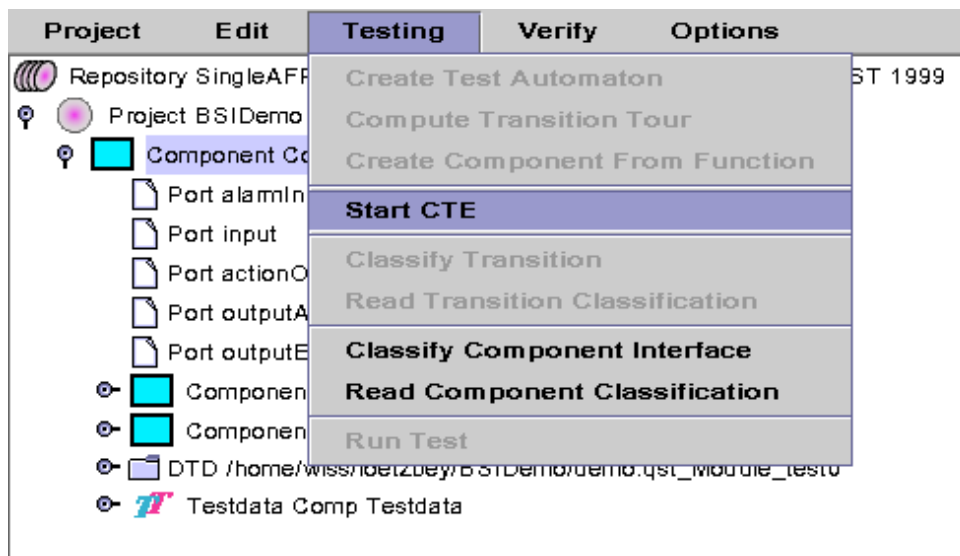


Figure 26: Starting the CTE

It is also possible to do the steps one by one. Step 1 is done by clicking on `Classify component interface`. This writes out the CTA information to a file. Then you can start the CTE and write the classification tree and test cases to an arbitrary CTE file. With `Read Component Classification` you can read back the test cases and generate the test automaton. Note that the appropriate component must be marked to read back the test cases from the CTE file.

The generated test automaton consists of a single state with one transition for each test case. The transitions contain only the input values or the classes of input values determined by the CTE. Concrete input data can be given in a later step (see Section 9.2.5). Of course it is not possible to derive the desired outputs from the component's interface. A reference implementation is needed to determine the outputs (see Section 9.2.5 and 9.3.2).

### 9.2.2 Test Basis: SSD with STDs

If the specification includes a behavior description, the test case generation should include the given behavioral aspects in the generated test cases. We will differentiate the following two scenarios:

- one component with associated automaton (unit test)
- several communicating components with associated automata (integration test)

**Unit Test** The unit test tests a single component of the system. Test cases are derived from the behavioral description of the component, i.e. the associated STD. Arbitrary STDs have not only a control state but also a data state. This makes the test sequence generation more complicated. If the data state of the automaton influences the change of the control state, i.e. influences the ability of some transitions to fire, not all transition sequences are feasible. Such automata should be transformed to simple automata where all transition sequences are feasible. Otherwise the test sequentialization produces transition sequences which have no corresponding trace of input/output values that is executable on the test object and on the specification.

The Quest development method provides the combined function and trace test (CFTT) [Sad98, Sad97] strategy for deriving suitable test cases for single components. This strategy includes a data partition that transfers a part of the data state into the control state of the automaton. Actually this task is not supported by the AUTOFOCUS/Quest tool and has to be done manually, but methodical support is provided in analysis paper of the quest project ([SK97]).

**Integration Test** The objective of the integration test is the detection of faults in the interaction of two or more previously tested components. AUTOFOCUS/Quest supports the integration test with the channel test. The channel test finds errors on a specific channel between two components. Possible sources of errors can be wrongly interconnected components or different implementations of used protocols. Actually AUTOFOCUS/Quest provides methodical support for generating a test automaton that fits the needs for the channel test. For details see analysis paper of quest ([SK97]).

**Classifying Transitions** One of the core AUTOFOCUS/Quest benefits is the creation of a classification proposal of the input data space of certain transitions. These proposed classifications are imported to the CTE-tool. The CTE-tool

enables the tester to partition the input data according to the classification tree method ([[GWG95](#)]). As long as the tester sticks to some conventions, the marked tests can be reimported to the AUTOFOCUS/Quest tool.

Alike the classification of component interface, the classification of transitions can be either done in one guided procedure or three single steps:

- STEP 1: produce CTA-File from single transition and data type definitions
- STEP 2: start CTE with generated CTA-File and save test cases in CTE-File
- STEP 3: read CTE-File and insert new transitions in automaton

The guided procedure is started with `startCTE` in the `Testing` menu. Note that a transition node has to be selected because `startCTE` is sensitiv with regard to the selected node. If a transition node is selected, `startCTE` will start the procedure to classify the input data of a transition. If a component node is selected, `startCTE` will classify the component interface. If you prefer to do the steps one by one, step 1 is initiated by pressing on `Testing->Classify Transition` and step 3 is done with `Testing->Read Transition Classification`. Be aware that the same transition is selected in the browser when doing step 1 and 3. Otherwise, the test cases cannot be reimported to AUTOFOCUS/Quest correctly.

### 9.2.3 Test Basis: Functions

Function definitions are the third test basis. By selecting a function definition and executing `Testing->Create Component from Function` AUTOFOCUS/Quest creates a new component in the `Testdata` folder. The new component has the same name and interface as the function definition and its associated automaton consists of one single state and a loop transition for each defining equation. The automaton reflects the behavior of the function definition except for the sequence of the equations resp. transitions. In function definitions the order is significant whereas the choice fo the transitions in an automaton may be non deterministic.

### 9.2.4 Deriving sequences from test automata

For the derivation of suitable test sequences AUTOFOCUS/Quest provides the transition tour algorithm. The transition tour algorithm computes a path of minimal length that covers all transitions of the automaton. Note that the automaton must

be strongly connected, i.e. all states must be reachable from each state, otherwise the algorithm fails. In order to start the transition tour algorithm, just mark the desired automaton and click on `Testing->Compute Transition Tour`. If the automaton is strongly connected, the transition tour algorithm produces an EET that represents a test which covers all transition of the automaton.

### 9.2.5 Test Basis: EETs

All tests are stored as EETs. Therefore it is possible to use manually created EETs for testing. This is especially useful for the system test. EETs from use cases are a good source for creating tests for the system test. Beyond that, regression testing can be realized by using protocols from former test runs as inputs.

## 9.3 Performing Tests

In order to perform a test an implementing class-file must be assigned to the component. The class-file must fulfill the conventions determined in Sect. 9.3.1. Then you can run the test as described in Sect. 9.3.2. Sect. 9.3.3 gives some hints how to interpret the results of the test run.

### 9.3.1 The Test Object Interface

see Reference Manual

### 9.3.2 Running the Test

Before running a test you have to associate a class file with the component under test. Therefore mark the component and click on `Edit->Create Subnode`. Then choose `Code` and enter the file name of the class. When determining the file name, use slashes (`/`) as delimiter and omit the ending `.class`. Example:

```
/home/QuestUser/ClassesUnderTest/Example
```

Then you can mark an arbitrary EET that is associated with the component and start the test with `Testing->Run Test`. The test driver will now load the class into the virtual machine and try to execute the I/O sequence that is contained in the EET. Note that boxes and loops are not yet supported by the test driver. After the test is finished, a protocol of the test run will be generated and stored as an EET of the component under test.

### 9.3.3 Interpreting the Results

There may be three possible types of errors in the protocol EET:

- `WRONG_VALUE_ERROR`: Message has a wrong value.
- `UNEXPECTED_MESSAGE_ERROR`: This is an additional message and does not exist in the original test.
- `NOT_RECEIVED_ERROR`: The test driver expected this message but did not receive it.

## A Installation

In this section the requirements and the installation for AUTOFOCUS and Quest are described. While a freely available version from AUTOFOCUS is downloadable from the AUTOFOCUS home-page (<http://autofocus.informatik.tu-muenchen.de>), Quest isn't public available. Both tools are provided in zip archives and have a self installable archive included. See the readme file contained in the archive for latest informations.

### A.1 System Requirements

Both AUTOFOCUS and the Quest extensions run under Unix platforms and Windows NT. AUTOFOCUS is a freely available case-tool prototype. Requirements for AUTOFOCUS are:

- an Unix-like platform or a PC running Windows NT. The operating system must be capable to run Java 2. AUTOFOCUS is tested with Solaris 2.6, Solaris 7, Linux (Kernel 2.2) and Windows NT 4.0.
- a run-time version of Java 2. AUTOFOCUS is tested with the Sun version of Java 2 (JDK 1.2.2).
- GNU RCS (Revision Control System) version 5.0 or higher, as the repository server uses RCS to provide access and version control. AUTOFOCUS is tested with GNU RCS version 5.7.
- an Unix-like `rm` commando. Of course this is available for all Unix platforms and only needed for Windows NT.

- an unzip commando to unpack the archive containing AUTOFOCUS.
- to connect the sample multimedia frontends provided with the distribution (the traffic lights example and the elevator example) to AUTOFOCUS' simulation environment SIMCENTER, you need a PC running Windows NT and the multimedia tool "FormulaGraphics".

A self installable version of AUTOFOCUS is available from the AUTOFOCUS home-page (<http://autofocus.informatik.tu-muenchen.de/index-e.html>). Note that you have to register and to accept our license agreement, which can be done online, to download AUTOFOCUS. Compiled versions of the GNU RCS 5.7 and of an Unix-like rm commando for Windows NT are available from the AUTOFOCUS-download page.

The Quest-extensions are not freely available, so there is no public available download page. Requirement for the Quest-extensions are:

- preferable an Unix-like platform or with some shortcomings a PC running Windows NT. The operating system must be capable to run Java 2. The Quest-extensions are tested with Solaris 2.6, Solaris 7, Linux (Kernel 2.2) and Windows NT 4.0. For the connection to VSE Solaris 2.5 (or later) is needed. For the connection to CTE Solaris 2.5 (or later) or Linux (Kernel 2.2) is needed.
- a run-time version of Java 2. The Quest-extensions are tested with the Sun version of Java 2 (JDK 1.2.2).
- an unzip commando to unpack the archive containing Quest.

Optional requirements are:

- for the connection to the theorem prover: VSE II version 1.02c or later.
- for the connection to the model checkers:
  - SMV version 2.5
  - SATO version 3.2
- for the testing environment: CTE



## A.2 Installation

For further notes on installation and configuration of AUTOFOCUS see the readme file contained in the zip archive.

For the installation of the Quest tools follow these steps:

1. Unpack the archive quest.zip with your favorite unzipper to a temporary directory. E.g. on Unix-like platforms:

```
cd /tmp
unzip quest.zip
```

This will create a directory Quest-Install with two files: Readme.txt and install.class.

2. Please readme file for up to date instructions.
3. Change to the directory Quest-Install and start the installer<sup>15</sup>:

```
cd Quest-Install
java install
```

4. After a while you should see the “Welcome to the Quest Setup program” (see Fig. 27).
5. The steps through the Quest setup should be self explanatory. The main thing is, that you have to enter a directory in which the Quest tools will be installed (see Fig. 28). That directory is the Quest-Home.
6. After the installation process the Quest tools reside in Quest-Home/bin. It is recommended, that you expand your path-settings appropriate.

```
export PATH=Quest-Home/bin:$PATH
```

A file named .quest with environmentvariable settings will be created in the Quest-Home directory and if Quest setup is not started as a user with root privileges, then it will be copied to the installers home directory. This configuration file is needed for the Quest tools.

---

<sup>15</sup>On Unix systems the installer stores some information in the directory /bin or /usr/bin, e.g. an uninstall program.

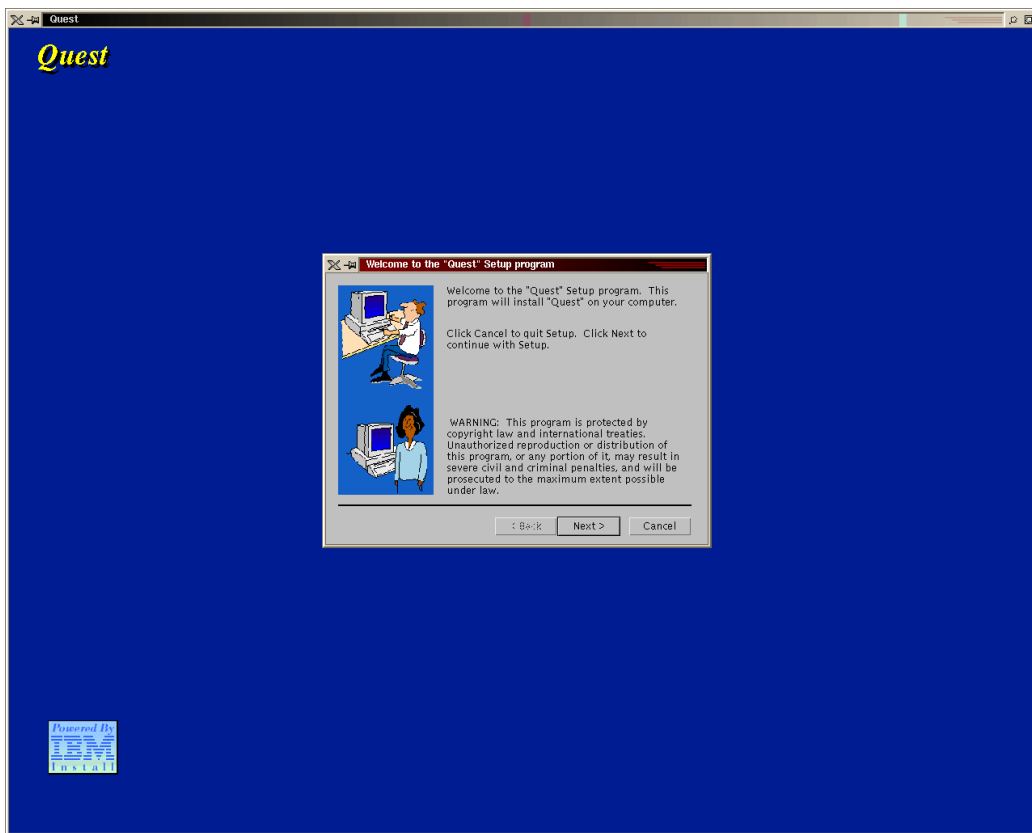


Figure 27: The welcome screen of the installer

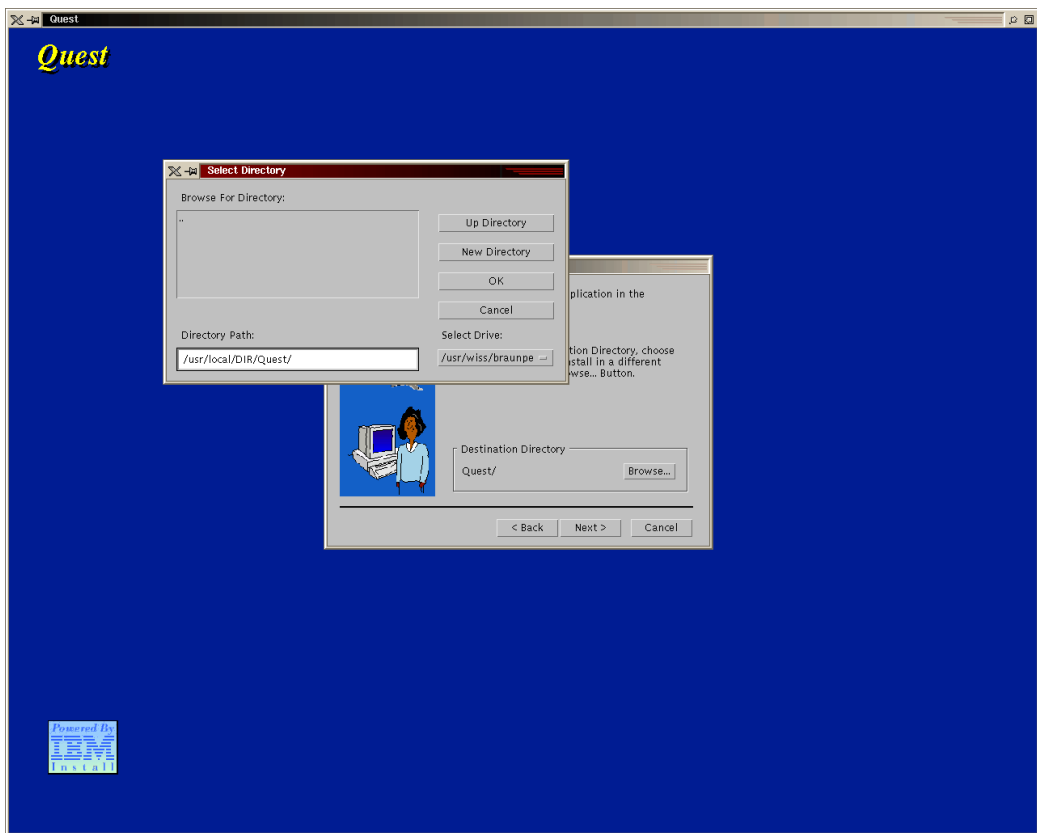


Figure 28: Choose a destination directory

7. If not already done, the other programs like VSE II, SMV, CTE, needed for the Quest tools, have to be installed now and should be added to your path-settings.
8. If another user wants to run the Quest tools, she/he should expand her/his path-settings appropriate and she/he have to copy the file `.quest` to her/his home directory.

### **A.3 Configuration**

Normally no further configuration of the Quest tools are needed. All settings are done by the Quest setup program. All user configurable settings are stored in the file `/ .quest`.

### **A.4 Uninstallation**

To uninstall the Quest tools on Unix platforms follow those steps:

1. Log in as the user who installed the Quest tools.
2. Start the uninstall program:

```
juninst Quest-Home/UnInst
```

3. Follow the suggested steps.

After this you may remove `juninst` from your `/bin` directory.

To uninstall the Quest tools on Windows platforms go to the Add/Remove Programs dialog in the Windows Control Panel and remove the program.

## **B Case Studies and Examples**

In this section we present the examples / case studies that have been modelled and checked within the project Quest. These examples are part of the distribution and can be used to test the tool. There are three case studies, each of them has some examples. Further work on the case studies is mentioned in Appendix [B.4](#). To run an example the QML representation has to be loaded from the `Examples` directory of Quest.

1. Storm Surge Barrier: A model of the emergency closing system of storm surge barrier in the Oosterschelde. The presented models have been used in 1998 to improve the specification of the system, which now has evolved further.
  - StormSurgeBarrier/SSB Version without initialization (close to original specification)
  - StormSurgeBarrier/QSS a version with initialization (like the running FriscoF simulation)
  - StormSurgeBarrier/Comparator two models of a comparator with different types

See Appendix [B.1](#) for a description of this case study.

2. The Banking System (presented at the FM99) with two examples:
  - The complete model FM99/FM99 contains the complete specification
  - The abstract model FM99/BankAbs contains a version with only single valued types. This is an ad hoc abstraction of the model and can be used to model check the complete system.

See Appendix [B.2](#) for a description of this case study.

3. A Traffic Light Control System: The old AUTOFOCUS example of the traffic light system for a pedestrian crossing over a street [[HMS+98](#)] has been re-modelled using the expressive power of functional data types.
  - TrafficLights/QuestTL contains the model.

The case studies are stored within the QuestML format and can be loaded into AUTOFOCUS and into the model browser. Furthermore there are some properties available, that can be loaded into the model browser to check something. The files are located in the `Examples`-directory (see installation [A](#)), and can be loaded into the model browser as described in Section [4.2](#). Figure [29](#) shows the sub-directories of the `Examples` directory, each containing a case study:

## **B.1 Emergency Closing System of Storm Surge Barrier**

The emergency closing system of the storm surge barrier in the Oosterschelde is the biggest case study. It has been analyzed with different methods and several models have been build. In the directory `StormSurgeBarrier` there are the following files:

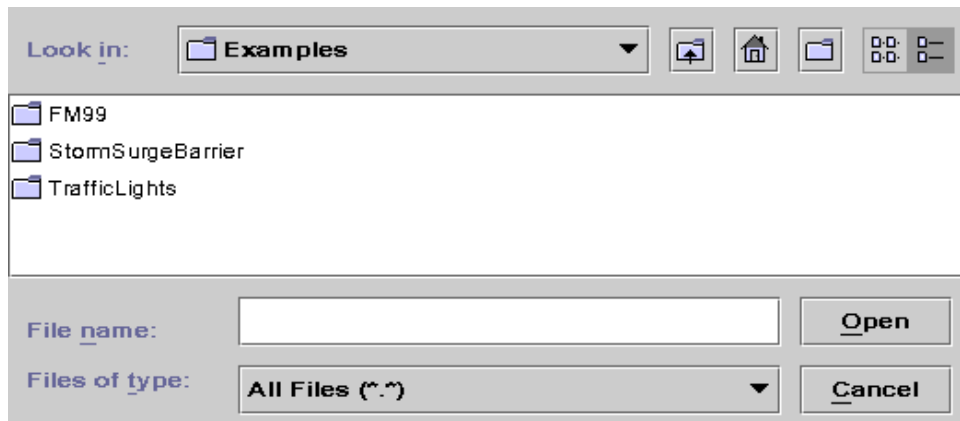


Figure 29: Content of the Examples directory

- SSB: the original model, see Section [B.1.1](#)
- QSS: an improved model, see Section [B.1.2](#)
- Comparator: a comparator for sensor signals, see Section [B.1.3](#)

Section [B.4](#) concludes with references to further work.

Starting point of our activities was an internal paper of the Dutch SIMTECH company [[vdMvdW98](#)] that should be the basis for a public offer for building a new emergency closing system, because the old system (running on a PDP11) is outdated. However, since several flaws have been found, and due to several other reasons, the emergency closing system is designed in a different way. Therefore the case study, (in the presented version) can be published (see [[vdMC99](#)]).

### B.1.1 The Original Model

The original model is contained within the file SSB. The model correspond to the original specification from the SIMTECH company in [[vdMvdW98](#)]. It contains the unexpected behaviour, that at the system is started, the output of the channel `notCLOSE` is not present. This denotes that the close signal is given and the barrier closes. With an initialization phase, this unexpected behaviour could be avoided. This unexpected behaviour was found during simulation of the system. Furthermore the model contains the ‘famous’ [[vdMC99](#)] bug in the statemachine that can be found using model checking.

### B.1.2 An Improved Model

The improved model is contained within the file `QSS`. In this model we integrated an initialization phase and fixed the bug of the statemachine using a boolean variable `closeHappened` to store whether a close signal has occurred in the system run. This model has been described very detailed in [Gie99].

### B.1.3 A Comparator for Sensor Signals

To illustrate the abstraction mechanisms and the generation of proof obligations for the correctness of the abstraction functions a comparator for sensor signal has been modelled. The abstract model of the comparator is identical to the comparator modelled within the above models of the storm surge barrier. Using abstractions and the verified abstract model we can insure the correctness of the concrete model. The correctness proofs have not been carried out using VSE, due to limited resources within the project. However similar proofs have been carried out using the Isabelle system [Ham99]. The essence of this work has been that it is much easier to discharge the proof obligations, compared with the finding of correct (and useful) abstractions. To ease this process we integrated the abstraction chooser within our framework.

The comparators are described within Section 7.2, the generated correctness conditions can be found in Appendix C.

## B.2 FM99 Banking System

On the Formal Method World Congress 1999 in Toulouse there was a tool exhibition with a competition. Every participating tool was invited to model a banking system with tills and transactions to withdraw money from accounts. Critical properties had to be formulated and verified. We participated with our tool and modelled the banking system. Due to the simple modelling language and the rich possibilities to verify properties our tool AUTOFOCUS with Quest won the first price.

Descriptions of the requirement document, and a paper describing our solution can be found under <http://autofocus.in.tum.de>. More information on the formal method world conference and the competition can be found in <http://www.fmse.cs.reading.ac.uk/fm99/>.

In the `Examples` directory there is a subdirectory `FM99` containing

- FM99 the complete model in QuestML format

- BankAbs an abstract model in QuestML format

Both Files contain some critical properties of the system.

### **B.2.1 Complete Model**

We modeled the banking system with different hierarchic graphical description techniques for structure and behavior. The model is based on a finite, functional description of datatypes for example for credit cards and operations likes `max-ForToday`.

The model includes two tills (user interaction, simple checks, communication with the system), two lossy connections, and a central including drivers (for the lossy connections) and a database. We did not model the database in detail, but we used nondeterminism of `AUTOFOCUS` to decide whether money can be withdrawn or not. This level of abstraction allowed to prove many interesting properties, especially on the lossy connections and the drivers.

The model is hierarchic, reuses several state transition diagrams (for example both tills have the same behaviour), and it is based on a rich set of data definitions describing the information on credit cards and transactions. This made the model very complex, such that it could not be model checked completely, but bounded model checking could be applied. To model check the complete system we built an abstraction (see Section [B.2.2](#)).

### **B.2.2 Abstract Model**

The file `BankAbs` contains an abstraction of the system that has been generated by replacing all data types of the system by single valued types. For example a the new type `Card` for credit cards contains only the element `AnyCard`. Furthermore the file contains two properties of the complete system, stating that the connection between the tills and the central is working, i.e. if the till sends any message, sometimes there will be a response. Of course in this model no statements concerning the content of the messages hold, however model checking the complete system only takes about one minute (and one minute for the generation).

## **B.3 Traffic Light Control System**

The traffic light control system is the smallest case study in the `Examples` directory. It contains the following files:



- `QuestTL`: The model in QuestML Format (including the DTD-Files `QuestTL_Module_Signal`, `QuestTL_Module_Lights`, `QuestTL_Module_Keys`, `QuestTL_Module_TimeConstants`)
- `Timer.prop`: a true property of the timer
- `System.prop`: a true property of the system

The model can be used for demonstrations, because it is very small (fast) and understandable. The model describes a small traffic light system, similar to the traffic light system described in [HMS<sup>+</sup>98]. However, it has more components (for example a timer) and it uses functional datatypes for the Lights. This reduces complexity of the state transition diagrams and increases understandability of the model. The model contains several “unexpected behaviours”, that can be found out using model checking, for example the timer does not necessarily produce timeouts, if it has been set. The reason (counter example) is that it can be set again before it produces the timeout. In this case the timer starts again. In the complete model however, this situation does not occur, since the timer is used correctly.

The timer can also be used to benchmark model checking, because there are (in the File `QuestTL_Module_TimeConstants`) constants that determine the length of the phases. Setting them to high values and increasing the `MaxInt` value in the browser can lead every system to its limit.

## B.4 Other Work

In this section we refer to further work (not contained in the `Examples` directory) that has been done using the tools and methods of Quest.

- The internal paper “Testgenerierung in Quest” from S. Sadeghipour and T. Klein describes the test methods of Quest, especially how the communication test method has been applied to test communication channels of the storm surge barrier model.
- Within the analysis of the storm surge barrier several models have been built. The `AUTOFOCUS/Quest` models have been described within the previous sections. Another model has been built using a direct encoding into VSE. The model has been build such that it allows for the specification of real time properties, and several critical aspects have been verified. The work is described in the internal paper “Modelling, Specification and Verification of an Emergency Closing System” by W. Stephan, G. Roch and M. Brodski.

- The Adelard company has verified the design of the storm surge barrier [vdMC99]. Some parts have been modelled with HOL, other parts have been encoded directly into the SMV system.

## C Correctness of Abstractions

This section contains all proof obligations that are generated for the correctness of the abstraction defined in Section 7.2.

### C.1 Definition of the Abstraction Function

Since the generation of proof obligation involves the application of the abstraction function, it is not necessary to explicitly define (in terms of VSE II) the abstraction for all elements of the model (Port, Types etc.), however since the relation between the states is essential the abstraction function has to be defined for states. In the example this is:

```

/* specification of state abstractions */
THEORY OpenBarrier_AbsState_Definition
  USING
    Concrete_States ;
    Abstract_States

  FUNCTIONS
    AbsStates : tConcrete_States -> tAbstract_States

  PREDICATES
    AbsInitial : tAbstract_States ;
    ConcInitial : tConcrete_States

  VARS  Abs      : tAbstract_States ;
        Conc     : tConcrete_States

  AXIOMS

    FOR AbsInitial : DEFPRED AbsInitial(Abs) <->
      IF Abs = Abstract_States.sMAIN THEN TRUE
      ELSE FALSE FI
    FOR ConcInitial : DEFPRED ConcInitial(Conc) <->
      IF Conc = Concrete_States.sMain THEN TRUE

```

```

    ELSE FALSE FI
  FOR AbsStates : DEFFUNC AbsStates(Conc) =
    IF Conc = Concrete_States.sMain
    THEN Abstract_States.sMAIN
    FI
  THEORYEND

```

## C.2 Homomorphosm Proof Obligation

To express the proof obligation that ensures that the abstraction function is an homomorphosm within VSE II we use the follwoing theory:

```

/* correctness of the abstraction function */
TLSPEC OpenBarrier_AbsState_Correctness
  INCLUDE Conc =Concrete ;
  Abs = Abstract

  SATISFIES OpenBarrier_AbsState_Property
TLSPECEND

```

The important proof obligation is in the follwoing theory.

```

TLSPEC OpenBarrier_AbsState_Property
  USING   OpenBarrier_AbsState_Definition
  INCLUDE Conc =Concrete ;
  Abs = Abstract

  VARS   Conc : tConcrete_States ;
  Vl : Int

  SPEC
  [] ALL Conc :
    ConcInitial(Conc) -> AbsInitial(AbsStates(Conc)) ;
  /* correctness for transition
    ((owl + openDifference) < iwl) ;
    IntIWL?iwl ;
    IntOWL?owl ;
    OPENDif!Present; */
  [] ALL Vl :
    <> (Conc.AFi_ControlState = Conc.Concrete_States.sMain
      AND Vl = Conc.l) ->
    ((Conc.AFi_ControlState = Conc.Concrete_States.sMain AND

```

```

Channel_Int.ChannelBase.is_Msg( IntIWL ) AND
Channel_Int.ChannelBase.is_Msg( IntOWL ) AND
( ( Channel_Int.ChannelBase.Val( IntOWL ) + openDif-
ference )
      < Channel_Int.ChannelBase.Val( IntIWL ) ) AND
Conc.AFi_ControlState' = Conc.Concrete_States.sMain
) ->
(AbsStates(Conc.AFi_ControlState) =
  AbsStates(Conc.Concrete_States.sMain) AND
  Channel_SensorSig.ChannelBase.is_Msg( IWL ) AND
  Channel_SensorSig.ChannelBase.is_Msg( OWL ) AND
  ( add( Channel_SensorSig.ChannelBase.Val( OWL ) , S00 )
      << Channel_SensorSig.ChannelBase.Val( IWL ) ) AND
  AbsStates(Conc.AFi_ControlState') =
    AbsStates(Conc.Concrete_States.sMain)
)
) ;

/* correctness for idle transition in state Main */
[] ALL V1 :
<> (Conc.AFi_ControlState = Conc.Concrete_States.sMain
    AND V1 = Conc.l) ->
((Conc.AFi_ControlState = Conc.Concrete_States.sMain AND
  NOT(Channel_Int.ChannelBase.is_Msg( IntIWL ) AND
    Channel_Int.ChannelBase.is_Msg( IntOWL ) AND
    ((Channel_Int.ChannelBase.Val( IntOWL )
      + openDifference )
      < Channel_Int.ChannelBase.Val( IntIWL ))
    ) AND
  Conc.AFi_ControlState' =
    Conc.Concrete_States.sMain
) ->
(AbsStates(Conc.AFi_ControlState) =
  AbsStates(Conc.Concrete_States.sMain) AND
  NOT(Channel_SensorSig.ChannelBase.is_Msg( IWL ) AND
    Channel_SensorSig.ChannelBase.is_Msg( OWL ) AND
    ( add( Channel_SensorSig.ChannelBase.Val( OWL ) , S00 )
      << Channel_SensorSig.ChannelBase.Val( IWL ) )
    ) AND
  AbsStates(Conc.AFi_ControlState') =
    AbsStates(Conc.Concrete_States.sMain)
)
)
)

```

TLSPECEND

### C.3 Strengthening Proof Obligation

The strengthening proof obligation defines the mappings between the concrete and the abstract types in the following theory:

```
THEORY OpenBarrier_Strengthen_Definition
  USING   OpenBarrier_AbsState_Definition ;
          MAINComparator_Module_HWSignal

  FUNCTIONS

          Int2SensorSig : SensorSig -> SensorSig;
          Signal2Signal : Signal -> Signal

  VARS
/* variables for function Int2SensorSig */
  V1Int2SensorSig : Int;
/* variables for function Signal2Signal */
  V1Signal2Signal : Signal;
  cInt : Int ;
  cSignal : Signal

  AXIOMS

/* axiom for the function Int2SensorSig :
fun Int2SensorSig(x) = S00;
*/
  FOR Int2SensorSig :
    DEFFUNC Int2SensorSig(V1Int2SensorSig) = S00

/* axiom for the function Signal2Signal :
fun Signal2Signal(x) = x;
*/
  FOR Signal2Signal :
    DEFFUNC Signal2Signal(V1Signal2Signal) =
      V1Signal2Signal

THEORYEND
```

The correctness condition of the abstraction function with respect to the selected properties (the strengthening) is

```

THEORY OpenBarrier_Strengthen_Property
  USING   OpenBarrier_Strengthen_Definition
  VARS    IntIWL : Int ;
          IntOWL : Int ;
          l : Int

  AXIOMS

/* Strengthening for Property:
  Concrete:
    [ ]((Val(Comparator.Combine.Concrete.IntOWL)
      < Val(Comparator.Combine.Concrete.IntIWL)) =>
      <>((Comparator.Combine.Concrete.OPENDif ! Present))))
  Abstract:
    [ ]((Val(Comparator.Combine.Abstract.OWL)
      << Val(Comparator.Combine.Abstract.IWL)) =>
      <>((Comparator.Combine.Abstract.OPENOK ! Present))))
*/
( ( Int2SensorSig( IntOWL ) << Int2SensorSig( IntIWL ) )
  -> ( IntOWL < IntIWL ) )
AND
( ( Signal2Signal( OPENDif ) = Present )
  -> ( OPENDif = Present ) )

THEORYEND

```

again the fact that this condition is true is expressed in a theory with a SATISFIES section:

```

THEORY OpenBarrier_Strengthen_Correct
  USING   OpenBarrier_Strengthen_Definition
  SATISFIES
          OpenBarrier_Strengthen_Property

THEORYEND

```

## References

- [BLSS00] P. Braun, H. Lötzbeyer, B. Schätz, and O. Slotosch. Consistent Integration of Formal Methods. In *Tools for the Analysis of Correct Systems (TACAS)*, 2000. to appear.

- [BS99] M. Broy and O. Slotosch. Enriching the Software Development Process by Formal Methods. In *Current Trends in Applied Formal Methods 98*, LNCS 1641, 1999.
- [CGL92] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *Proc. 19th ACM Symp. Principles of Programming Languages*, pages 343–354. ACM Press, 1992.
- [DGG97] Dennis Dams, Orna Grumberg, and Rob Gerth. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):22–43, 1997.
- [Gie99] P. Gierl. Casestudy: The Oostershelde NSS, 1999. Fortgeschrittenenpraktikum, Technische Universität München.
- [GL93] Susanne Graf and Claire Loiseaux. A tool for symbolic program verification and abstraction. In C. Courcoubetis, editor, *3th Int. Conf. on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 71–84. Springer-Verlag, 1993.
- [GWG95] M. Grochtmann, J. Wegner, and K. Grimm. Test Case Design Using Classification Trees and the Classification-Tree Editor. In *Proceedings of 8th International Quality Week, San Francisco*, pages Paper 4–A–4, May 30–June 2 1995.
- [Ham98] Tobias Hamberger. Verifikation einer Hub-schrauberüberwachungskomponente mit Isabelle und STeP, 1998. Student software development project, Technische Universität München, Germany.
- [Ham99] Tobias Hamberger. Kombination von Theorembeweisen und Model Checking für I/O Automaten. Master’s thesis, Computer Science Department, Technical University Munich, 1999.
- [HEea96] F. Huber, G. Einert, and et al. *AutoFocus User’s Manual*, 1996.
- [HMR<sup>+</sup>98] F. Huber, S. Molterer, A. Rausch, B. Schätz, M. Sihling, and O. Slotosch. Tool supported Specification and Simulation of Distributed Systems. In *International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 155–164, 1998.

- [HMS<sup>+</sup>98] F. Huber, S. Molterer, B. Schätz, O. Slotosch, and A. Vilbig. Traffic Lights - An AutoFocus Case Study. In *1998 International Conference on Application of Concurrency to System Design*, pages 282–294. IEEE Computer Society, 1998.
- [HS96] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In M. Gaudel and J. Woodcock, editors, *Formal Methods Europe*, Lecture Notes in Computer Science, pages 662–681. Springer-Verlag, 1996.
- [HS97] F. Huber and B. Schätz. Rapid Prototyping with AutoFocus. In A. Wolisz, I. Schieferdecker, and A. Rennoch, editors, *Formale Beschreibungstechniken für verteilte Systeme, GI/ITG Fachgespräch 1997*, pp. 343-352. GMD Verlag (St. Augustin), 1997.
- [HSE97] F. Huber, B. Schätz, and G. Einert. Consistent graphical specification of distributed systems. In *FME '97: 4th International Symposium of Formal Methods Europe, LNCS 1313*, pages 122 – 141, 1997.
- [Jon93] M. P. Jones. *An Introduction to Gofer*, August 1993.
- [KBRS98] Koob, Baur, Reif, and Stephan. *VSE II Development Method*, 1998.
- [Kur87] R.P. Kurshan. Reducibility in analysis of coordination. In Kurzhanski Varaiya, editor, *Discrete Event Systems: Models and Applications*, volume 103 of *Lecture Notes in Control and Information Science*, pages 19–39. Springer-Verlag, 1987.
- [LGS<sup>+</sup>95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.
- [Mer97] Stephan Merz. Rules for abstraction. In R. K. Shyamasundar and K. Ueda, editors, *Advances in Computing Science—ASIAN'97*, volume 1345 of *Lecture Notes in Computer Science*, pages 32–45, Kathmandu, Nepal, December 1997. Springer-Verlag.
- [MN95] Olaf Müller and Tobias Nipkow. Combining model checking and deduction for I/O-automata. In *Proc. 1st Workshop Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1995.



- [Mül98a] Olaf Müller. *A Verification Environment for I/O-Automata Based on Formalized Meta-Theory*. PhD thesis, Institut für Informatik, Techn. Univ. München, 1998.
- [Mül98b] Olaf Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory*. PhD thesis, Institut für Informatik, Technische Universität München, 1998.
- [Pau91] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.
- [Pla97] T. Plasa. *VSE II Benutzerhandbuch Grundsystem*, 1997.
- [PS99] J. Philipps and O. Slotosch. The Quest for Correct Systems: Model Checking of Diagramms and Datatypes. In *Asia Pacific Software Engineering Conference 1999*, 1999. to appear.
- [RSW97] G. Rock, W. Stephan, and A. Wolpers. Tool Support for the Compositional Development of Distributed Systems. In *Proc. Formale Beschreibungstechniken für verteilte Systeme, GI/ITG-Fachgespräch*. GMD-Studien Nr. 315, ISBN: 3-88457-514-2, 1997.
- [Sad97] S. Sadeghipour. Teststrategien auf Basis erweiterter, endlicher Automaten, 1997. Internes Papier.
- [Sad98] S. Sadeghipour. *Testing Cyclic Software Components of Reactive Systems on the Basis of Formal Specifications*. PhD thesis, Technische Universität Berlin, Fachbereich Informatik, 1998.
- [SK97] S. Sadeghipour and T. Klein. Testmethoden für formale, graphische Spezifikationen, 1997. Internes Papier.
- [Slo98] O. Slotosch. Quest: Overview over the Project. In D. Hutter, W. Stephan, P Traverso, and M. Ullmann, editors, *Applied Formal Methods - FM-Trends 98*, pages 346–350. Springer LNCS 1641, 1998.
- [SLW95] B. Steffen, K.G. Larsen, and C. Weise. A constraint oriented proof methodology based on modal transition systems. In *Proc. 1st Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 17–40. Springer-Verlag, 1995.

- [UBR<sup>+</sup>99] Ullmann, Baur, Reif, Siekmann, Scheer, and Moik. *Specification Language VSE SL Version 2*, 1999.
- [vdMC99] Meine van der Meulen and Tim Clement. Formal Methods in the Specification of the Emergency Closing System of the Eastern Scheld Storm Surge Barrier. In *Current Trends in Applied Formal Methods 98*, LNCS 1641, 1999.
- [vdMvdW98] Meine van der Meulen and K. van de Wetering. Requirements Specification NSS, October 19th 1998.
- [Wim00] G. Wimmel. Using SATO for the Generation of Input Values for Test Sequences. Master's thesis, Technische Universität München, 2000.
- [Wol86] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th ACM Symp. Principles of Programming Languages*, pages 184–193. ACM Press, 1986.
- [Zha97] H. Zhang. SATO: An efficient propositional prover. In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *LNAI*, pages 272–275, Berlin, July 13–17 1997. Springer.