TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Diplomarbeit

Specification Based Determination of Test Sequences in Embedded Systems

Bearbeiter:	Guido Wimmel
Aufgabensteller:	Prof. Dr. Manfred Broy
Betreuer:	Dr. Oscar Slotosch / Heiko Lötzbeyer
Abgabedatum:	15. Februar 2000

Ich versichere, daß ich diese Diplomarbeit selbständig verfaßt und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15.Februar 2000

. . . .
Guido Wimmel

Abstract

Finding input values for software tests is a difficult task, especially for distributed, embedded systems. In this work, we use the logical basis of AutoFocus system models to automate this and other important steps during software development.

In particular, we describe six possible ways to specify test cases. For these test cases, we show how to automatically determine input values and expected output values (a "test sequence") with the help of our logical model for AutoFocus system specifications, by solving a propositional formula using the tool SATO. One of these scenarios — specifying a test case with a formula in temporal logic — leads us to a bounded model checking algorithm for AutoFocus systems, in many cases an efficient alternative to conventional model checking to verify properties of the system. Our work also provides a connection of AutoFocus to the classical BDD-based model checker SMV.

Finally, the practicability of our approaches is demonstrated by exemplary performance figures for the different test cases.

Contents

List of Figures

Chapter 1

Introduction

1.1 Testing Embedded Systems

Computer systems are more and more invading all major parts of everyday life. Microcontrollers running specialized software revolutionized telecommunications industry by their use in mobile phones, increase safety and reduce fuel consumption of automobiles through intelligent motor control systems or anti-blocking technology, and make it possible for credit card owners to withdraw money almost all over the world at automated telling machines. But this is only the beginning: soon, computer technology will move into people's homes, controlling for example lights, heating, alarm systems, multimedia devices and household appliances, all connected via a common local network.

Such computer systems that are very closely integrated into products like automobiles or mobile phones are called *embedded systems*. The use of embedded systems for increasingly complex tasks and in distributed environments make embedded systems software development more and more demanding. Therefore, like it became necessary for conventional software many years ago, the techniques of software engineering also have to be applied in the design of embedded systems software.

It is an important aspect of software engineering to ensure the correctness of the developed systems. This is especially critical for embedded systems: whereas errors in conventional systems usually cost money, in embedded systems they can easily lead to the loss of human lives or even to major catastrophes (only consider the use of embedded systems in nuclear power plants or strategic missiles).

If the correctness of a system is crucial, formal verification techniques should be applied throughout the development process (e.g. using the verification support environment VSE [RSW97]). However, this is costly and time-consuming. Therefore, less critical systems are usually engineered with conventional approaches. The most common way to verify the correctness of a system in this case is *testing*. To be able to perform tests, we need a *specification* of the system that describes its required behaviour and was created in earlier steps of software development, usually in cooperation with the customer. The activity of testing is then to check whether the implementation is correct with respect to its specification.

Specifications can be merely informal, but to test safety-critical systems we need a formal specification describing the complete behaviour of the system. Such a specification can be created with the help of AUTOFOCUS, a CASE tool developed at Munich University of Technology and especially tailored to support the design of correct embedded systems software. In AUTOFOCUS, systems can be modelled as a collection of communicating extended finite automata, with a clean and simple underlying semantics.

A complete AUTOFOCUS specification describes every possible execution of the system. For systematic testing, the tester selects *test cases* that describe particular desired behaviours of the system (for example, each of the transitions in the automata should be executed at least once (a so-called *transition tour*), or a certain critical transition to be tested should be taken at some step in execution, or one of the automata should reach a certain control state, e.g. an error state). The execution of the specification described by a test case can then be compared with the actual behaviour of the implementation to check its correctness.

However, the implementation of an embedded system can be seen as a "black box", only being controlled by the input data we pass to it. How can we determine the input data required by the specification (and thus also by the implementation) so that a certain state is reached or a particular transition is executed ?

A solution to this important problem is given in this thesis. We show how to translate AUTOFOCUS system models into propositional logic and how to determine input data and the expected output data (a *test sequence*) for a test case from the specification, by solving a logical formula using the tool SATO.

There are many applications of our approach to translate AUTOFOCUS system models into logic, also outside the field of system testing. An additional result of our work is a bounded model checking algorithm for AutoFocus systems, in many cases an efficient alternative to conventional model checking to verify arbitrary properties of the specification. We also implemented a connection of AutoFocus to the classical BDD-based model checker SMV.

1.2 Structure of the Thesis

Our thesis is structured as follows:

- In the second chapter, we introduce the CASE tool AUTOFOCUS and explain how to use the description techniques it provides to specify embedded systems. As an example, we describe the specification of a cash point service.

- In the third chapter, we show how to translate an AUTOFOCUS system model into logic and how to calculate particular system executions and the corresponding input and output data as solutions to propositional formulas. We also describe how to translate AUTOFOCUS systems to SMV with the help of the logical model.

- In chapter 4, we explain how to use this approach for system testing and present six possible ways to specify test cases ("*test scenarios*", for instance the execution of a sequence of transitions in the specification or putting an automaton in the specification into a certain control state). One of these test scenarios leads us to the bounded model checking algorithm for AUTOFOCUS, which we will also present in this chapter.

- In addition, we created a Java implementation of the concepts and algorithms described in this thesis. The structure and usage of the implementation is explained in chapter 5. Our implementation was integrated into the AUTOFOCUS/QUEST project and can already be used in practice for test data determination, to translate a system specification to SMV or to perform bounded model checking.

- In chapter 6, we present performance statistics obtained with our implementation. With the help of these statistics, we give a first evaluation of the applicability of our methods for various test scenarios. We also include a comparison of the bounded model checking algorithm with conventional model checking using the BDD-based model checker SMV.

- Finally, the thesis concludes in chapter 7 with a summary and shows some of the limits of our approach and possible further work.

The appendix contains the complete AUTOFOCUS specification of the cash point example and a description of the tool SATO and its input language.

Chapter 2

Specifying Embedded Systems with AUTOFOCUS

The aim of this thesis is to show how testing can be integrated into embedded systems development by deriving test data directly from a system specification. But how do we specify a system in the first place ?

In this chapter, we present the CASE tool AUTOFOCUS, together with the Quest extensions. AUTOFOCUS [FMR+98, HMS+98] was developed recently at Munich University of Technology with the specific goal in mind to combine user-friendly graphical system design and support of simulation, code generation and formal verification of correctness. Unfortunately, none of the already established methods meets these requirements. For example, UML [BJR97] or SDL [IT93] provide powerful intuitive ways to describe a system, but defining their semantics precisely is a complex task, which prevents efficient formal verification. On the other hand, specification languages like Promela [HP96], CCS [Mil80] or SMV [McM93] lack user-friendly graphical description techniques and do not support early stages of system development well enough.

For these reasons, we used AUTOFOCUS as a basis for our work. AUTOFOCUS makes it possible to design systems in a hierarchical, view-oriented way — an approach that is very well established and facilitates its use in an industrial environment. However, it is also based on a well-founded formal background ("FOCUS", see [BDD+92]) and a fairly elementary underlying concept: communicating extended mealy machines. Thus, system complexity is limited enough to apply automatic verification algorithms. Unfortunately, we have to accept the drawback that this restricts our system specifications to quite basic, hardware-oriented ones — but this is by far compensated by the fact that we can prove their correctness !

We will now introduce the basic concepts of AUTOFOCUS by means of an example system, a cash-point service. This example is also used throughout the thesis to explain our specification-based system testing methods.

2.1 Informal Description of the Cash Point Service

The AUTOFOCUS specification of the cash point service has been presented by Dr. Oscar Slotosch at the 1999 Conference on Formal Methods in Toulouse. AUTOFOCUS was awarded the first prize, in a competition against over 30 other CASE tools. More information and a more detailed description of the cash point system can be found in [Slo99].

The general idea is that the cash point service consists of a number of tills that are connected via a communication link to a centrale. Each customer possesses a banking card. After inserting their card into the till and typing in the correct PIN (that is stored on the card), customers can view the balance of their accounts, withdraw money or ask for a statement of their accounts to be sent by post. The till contacts the central database to verify if withdrawals can be allowed. Cards can also be illegal, in which case they are kept by the till. The complete AUTOFOCUS specification of the cash point example can be found in appendix A.

2.2 System Structure Diagrams (SSDs)

Complex systems are usually modelled by dividing them into components that communicate with each other using a defined interface. This structural aspect of a system is covered in AUTOFOCUS by *System Structure Diagrams* (SSDs).

A SSD is similar to a data flow diagram and consists of a number of named components with input and output ports that are used to send and receive messages. The ports can be connected via channels, making it possible for components to exchange data. Ports that are not attached to a component in the system structure diagram represent the environment that can also generate and receive messages. Ports and Channels are assigned particular data types, and can in addition contain the empty value `NoVal` (see the explanation of data type definitions below). Thus, a system structure diagram describes both the syntactical interface (i.e. which kind of messages can be sent and received) and the topological structure of a system.

SSDs can be hierarchical, i.e. a component belonging to a SSD can have a substructure that is defined by a SSD itself. This way, a more detailed view of the system structure can be given. The ports attached to a component represent the environment for the SSD that refines this component. A hierarchical SSD can also be seen as a tree, considering its subcomponents and sub-subcomponents etc. At some stage, there will be components that can not be subdivided further, we will call them "*basic components*".

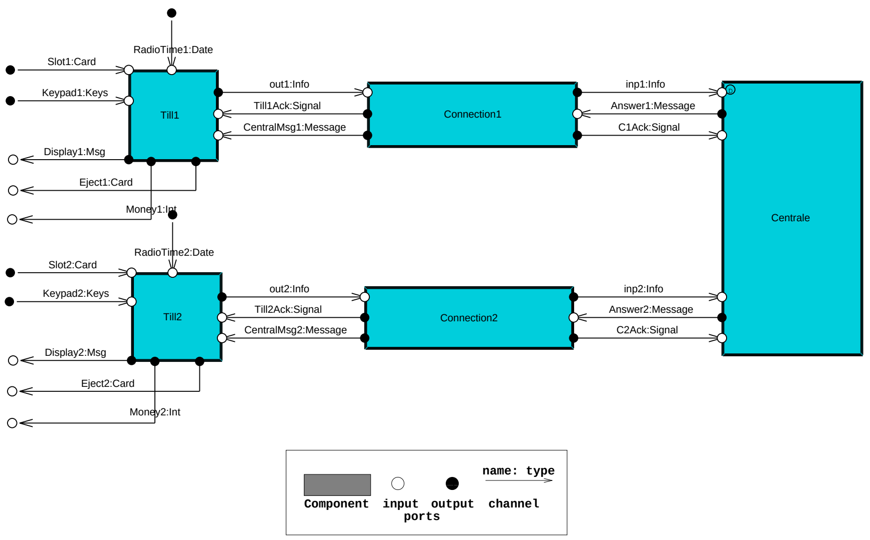


Figure 2.1: SSD for the Cash Point System

Fig. 2.1 shows the system structure diagram for the cash point service. It consists of two tills `Till1` and `Till2`, two `Connection` components that serve as a data link between the tills and the centrale, and the central data base itself (`Centrale`). Both `Till` components have ports to receive card data from the customer (`Slot1/2`) or terminal input (`Keypad1/2`), and can output information on a display (`Display1/2`) or issue money (`Money`). Via the channels between `Till1/2`, `Connection1/2` and `Centrale`, the tills exchange information with the central database.

Fig. 2.2 explains how the `Centrale` component is refined further into drivers for each connection (`Driver1/2`) and the actual processes (`Process1/2`) handling the requests from the tills.

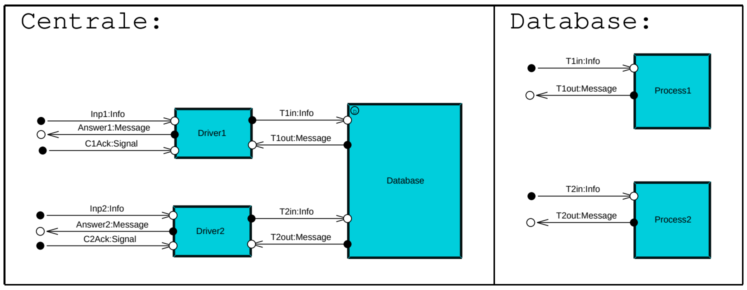


Figure 2.2: Refined SSDs for `Centrale` and `Database`

2.3 Local Variables for Components

A component can be associated with local variables using so-called *Component Data Declarations*. These simply specify a name, type and optionally an initial value for each local variable.

The `Till` components contain local variables for the data of the card recently inserted, the requested transaction and a timer t for timeouts. The corresponding data declarations are:

```
Card stored = Invalid
Info transaction = TA(Withdraw(0))
Int t = 0
```

(see the following section for an explanation of the used data types).

2.4 Data Type Definitions (DTDs)

Data types and functions used within AUTOFOCUS specifications can be defined with the functional language Quest [Slo98]. In addition to the integer type (`Int`), Quest allows the user to define complex hierarchical data types like the type "`Card`" storing data of a banking card or the type "`Info`" storing information about a requested transaction. Moreover, it is possible to specify user-defined functions containing pattern matching.

The user-defined data types in Quest are very similar to those in functional programming languages such as Gofer [Jon93]. For instance, the type `Card` is specified the following way, based on the types `Date` and `Cardinfo` (for the definition of the other data types in the cash point example, see its complete specification given in appendix A):

```
data Cardinfo = CI(PINCode: Int, accNo: Int);
data Date     = mkDate(Int);
data Card     = Invalid | Valid(info: Cardinfo,
                                dayAmount: Int, Date);
```

Thus, a variable of type `Card` can have the value "`Invalid`" (meaning the corresponding card is invalid), or the value "`Valid`(r, s, t)", where r, s and t are of type `Cardinfo`, `Int` and `Date` (meaning we have a valid card with card information, daily amount and a date stored on it). An example for a value for type `Card` would be `Valid(CI(1234,42),100,mkDate(17))`.

The general form of a data type declaration is

$$\begin{array}{ll} \mathit{data\ Type} = & C_1(sel_{11} : Type_{11}, \ldots, sel_{1n_1} : Type_{1n_1}) | \ldots | \\ & C_m(sel_{m1} : Type_{m1}, \ldots, sel_{mn_m} : Type_{mn_m}) \end{array}$$

.

Constructors, Discriminators and Selectors

The C_i (e.g. `Valid`) are called *constructors*, because they can be used to construct a value of the corresponding type (as for example via the *constructor term* `Valid`(r, s, t)). A constructor can have an arbitrary number of arguments, including no arguments at all. For each constructor there is also a *discriminator* `is_`$C_i(x)$ that evaluates to `True` if the value of x belongs to the alternative C_i (e.g. `is_Valid`(x) is `True` if x=`Valid`(...)).

The sel_{ij} are called *selectors*. If a value was generated by a constructor term that takes arguments, each selector returns one of the arguments. For example, `info(Valid(CI(1234,42),100,mkDate(17))) = CI(1234,42)`. If no names are given for the selectors, AUTOFOCUS automatically generates them by using the constructor names and a consecutive number (e.g. `mkDateSel1` is the selector to extract the day number from an expression of the type `Date`).

Data Type Definition for Boolean Values and Channels

The data types for Boolean values and Channels are defined in the same way:

```
data Bool = True | False;
data Channel(alpha) = NoVal | Msg(getVal: alpha);
```

The latter definition means that a channel of type `Channel(alpha)` can either be empty (denoted by `NoVal`) or carry a message of type `alpha` (`alpha` is a type parameter, and is replaced by the actual type of the channel, e.g. `Int` or `Card`).

User-Defined Functions

Finally, AUTOFOCUS data type declarations can contain *user-defined functions*. A user-defined function can be specified using different alternatives and pattern matching. The following example should clarify this:

```
data Action = SendMail | Withdraw(Int) | ViewBalance;
fun    criticalAction(Withdraw(amt)) = amt>500
     | criticalAction(SendMail)      = False
     | criticalAction(ViewBalance)   = False;
```

Thus, if x is of type `Action`, `criticalAction`(x) indicates the customer wants to withdraw more than 500 credits, which probably makes additional security measures necessary.

The meaning of such a definition should now be obvious. If the value of x is of the form `Withdraw(amt)`, the first alternative is taken, `amt` on the right side is replaced by its actual value, and `amt>500` is evaluated (for example, `criticalAction( Withdraw(600) ) = 600>500 = True`). If the value of x is of the form `SendMail` or `ViewBalance`, the result of `criticalAction` will be `False`.

2.5 State Transition Diagrams (STDs)

Up to now, we have only specified the structure and the interface of a system (i.e., which data types are used as input and output). The *behaviour* of particular components of a system can be described by *State Transition Diagrams* (STDs). The behaviour of a complete distributed system depicted by a certain system structure diagram is fully specified if each of its component has either a refinement or a behaviour.

STDs represent extended finite automata, meaning that in addition to the control state, they also have an internal state (the values of the local variables of the components). Each transition of the automaton has an annotation consisting of

- *Input patterns* of the form `i?Pattern`, where `i` is an input port. The corresponding transition is only executable if the value at the input port matches the pattern. If the pattern contains variables, they will be bound to their actual values and can be used later in the expressions for the precondition, output pattern and action (like in the function definitions shown in section 2.4).
  The input pattern "`i?`" is an abbreviation for "`i?NoVal`", specifying that the input port `i` should be empty. In input patterns of the form "`i?x`", `x` already refers the data of the message itself, i.e. we should think of it as really meaning "`i?Msg(x)`".
- A *precondition*, which is simply a boolean expression containing local variables and variables bound in the input patterns. For the transition to be executable, the precondition must evaluate to `True`.
- *Output patterns* of the form `o!expression`, determining which value will be written to the output port `o` when the transition is executed. If not all ports of a component are mentioned in the output patterns, the remaining ones will be set to the empty value `NoVal`.
- *Actions* of the form `local variable = expression`, determining to which value the corresponding local variable will be set.

STDs can also be hierarchical (i.e. a state can be refined by a STD itself). However, for our purposes, we require "flat" STDs, so we will not describe this feature. See the introductions on AUTOFOCUS [FMR+98] for details.

As an example, Fig. 2.3 shows the STD that describes the behaviour of the `Till` component. Focus on the transition (4) from state `CardEntered` to `Ready`, marked by

Precondition:	not(PINCode(info(storedCard)==pin))
Input pattern:	Keypad?PIN(pin)
Output patterns:	Eject!storedCard;Display!ErrorWrongPIN
Actions:	storedCard = Invalid

A card has just been entered into the system and its data was stored in the local variable `storedCard`. This transition is only executable if a PIN was entered with the keypad, i.e. the value at input port `Keypad` matches the pattern `PIN(pin)`. The actual value of the entered PIN (bound to the variable `pin`) is checked against the PIN contained on the stored card (see the precondition). If the PIN is incorrect, the transition is executed, the stored card is ejected, an error message is displayed and the local variable for the stored card is reset to `Invalid` so new cards can be inserted.

The STDs describing the behaviour of the other components of the cash point systems are shown in the appendix.

2.6 Extended Event Traces (EETs)

The final aspect of a system covered by AUTOFOCUS are event traces, describing exemplary runs by showing the components of a system and their communication with each other and the environment.

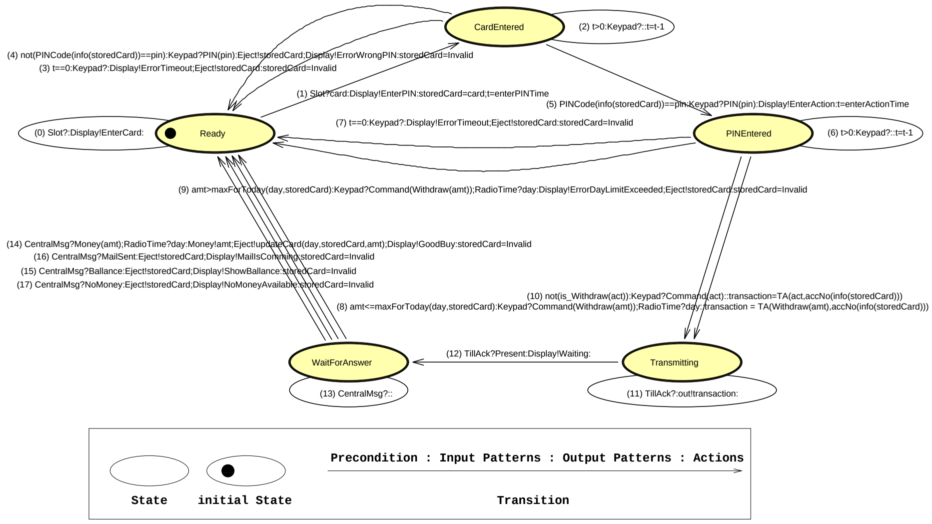


Figure 2.3: STD for the `Till` Component

Extended event traces (EETs) are related to the ITU-standardized message sequence charts (MSCs) [ITU96], and also support hierarchical concepts we cannot describe here in detail (see [FMR$^+$98]).

EETs can be used both at the beginning of system development (to specify required behaviour by use cases) and later to visualize simulation results, test cases or examples for erroneous behaviour ("counter examples" that show that a specificied system property is not fulfilled).

Fig. 2.4 shows an example for the communication behaviour of the `Till` component in the case mentioned above, where a wrong PIN is entered. The arrows denote messages, and the execution steps of the system are separated by dashed lines ("ticks").

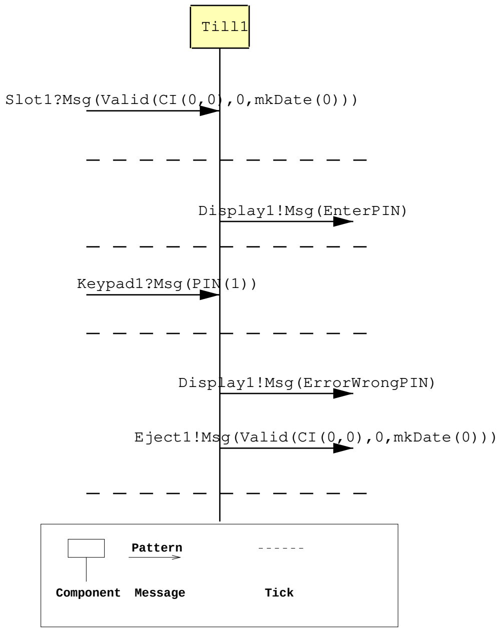


Figure 2.4: Example EET for the `Till` Component

Chapter 3

Logical Modelling of AutoFocus System Behaviour

In the last chapter, we have shown how to specify a system with AutoFocus by defining its structure (via SSDs) and the behaviour of its components (via STDs). If the specification is complete (i.e., each component has either a refinement or a behaviour), it can be used to describe the actual possible executions of a system.

In this chapter, we will show how to model the behaviour of a system specified with AutoFocus. The result will be a formula in propositional logic, the solutions of which describe all possible executions of the system up to a certain point in time (i.e., the first $k + 1$ execution steps). With the help of the tool SATO, we can then calculate a solution to the formula, corresponding to a particular possible execution of the system. Such an execution is given by the data at the input and output ports, the values of the local variables and the current control states of the automata at each step.

This lays the foundation for specification based test sequence determination: it is now easy to add additional predicates to the formula, so that a solution does not correspond to an arbitrary execution of the system, but shows us test data for a particular test case. This way, we can determine executions that put an automaton in the system into a certain state, that contain the execution of particular transitions etc. System testing using specification based test sequence determination is described in the following chapter.

We will also present an important intermediate result of the logical modelling of AutoFocus system behaviour: the translation of AutoFocus systems to the input language of the SMV model checker.

3.1 Semantics of AutoFocus System Models

To model the behaviour of an AutoFocus system, we need to know its semantics — i.e., what happens exactly when a transition in a state transition diagram is executed, how the components communicate with each other, etc. In this section, we give an operational semantics for AutoFocus system models by showing how to translate them into state transition graphs.

The graphical specification techniques of AutoFocus deliberately allow great flexibility in the definition of a semantics. In principle, all possible models of execution and communication (buffered, unbuffered, synchronous, asynchronous) could be employed. However, for our purposes we require a finite state space, because we later have to represent the system state using a finite number of boolean variables.

In this thesis, we will concentrate on a synchronous execution model using a global clock and message buffers of size 1. This execution model was chosen in the project

QUEST for the various simulation or verification tools that already exist (see e.g. [PS99, HS97]) because of the following reasons (see [HSE97]):

- It is the natural way of describing hardware oriented embedded systems, the main application area of AUTOFOCUS.
- The state space has a fixed size (when using finite data types), which facilitates the use of model checking techniques.

In particular, our definition of the semantics of an AUTOFOCUS model is based on the translation to the model checker SMV presented in [PS99]. Our approach can be seen as a generalisation of this translation, so as we will see, it can in fact also be used to translate AUTOFOCUS systems to SMV. Thus, we can later on compare our results gained with SATO with those calculated by SMV (see chapter 6).

3.1.1 State Transition Graphs

In our model, an AUTOFOCUS system executes in discrete steps, separated by the clock ticks. Thus, we can describe the operation of such a system by

- Its *global state*, i.e. the current data values on all channels, the values of the local variables of all the components and the current control states of the finite automata (STDs) assigned to the components.
- The *transition relation*, describing in which global states the system can be in the next clock cycle, as a result of executing transitions in the automata (depending on the current global state).

In addition, we need a set of *initial global states* that the execution of a system can begin with.

Formally, this description of the operational semantics corresponds to a so-called *state transition graph* (STG). A state transition graph is a triple (S, I, R), with the following elements:

- $S = d_1 \times d_2 \times \ldots \times d_n$, the global state space, consisting of n state variables which can be chosen from the domains $d_1, \ldots, d_n$.
- $I \subseteq S$, the set of initial states of the system.
- $T \subseteq S \times S$, the transition relation. If s is the current global state, then s' can be the global state in the next step if $(s, s') \in T$.

In the model given in this chapter, the types of the state variables can be arbitrary user-defined AUTOFOCUS data types. The set of initial states and the transition relation are given as boolean valued terms $I(s)$ and $T(s, s')$ containing the state variables, so that $I(s)$ evaluates to `True` if s is an initial state, and $T(s, s')$ evaluates to `True` if (s, s') is in the transition relation. In the term T for the transition relation, `next(x)` stands for the variable x', i.e. x in the next step. These terms can also make use of the constructor, selector and discriminator functions for the data types as explained in section 2.4. In sections 3.2 and 3.3, we will show how to eliminate user-defined types from the state space and represent it only by boolean variables.

As a motivation, Fig. 3.2 shows the STG for the very simple AUTOFOCUS system in Fig 3.1, containing only one component with two ports and an STD assigned to it. Both ports `i` and `o` can only take the values `NoVal` and `Msg(Present)`. The nodes of the corresponding state transition graph that gives the semantics for this system show its global state (in this case, determined only by the values on the ports, as the control state must always be Main). The arrows denote the transition relation and point to the possible states in the next clock cycle. In the initial states, the output port must be empty (denoted by NoVal), and the input port can have an arbitrary value. If the value at the input port is Msg(Present), the transition can be executed, and in the next state the value at the output port will be Msg(Present) (straight arrows). If the transition cannot be executed, the output port is cleared (value NoVal, dashed arrows). This is called "idle transition". Note that the initial states and the transition relation are actually not given explicitly as in the diagram, but as a boolean formulas. For the initial states, this would be

$I(s) = ((\text{state} == \texttt{Main}) \wedge \texttt{is\_NoVal(o)})$.

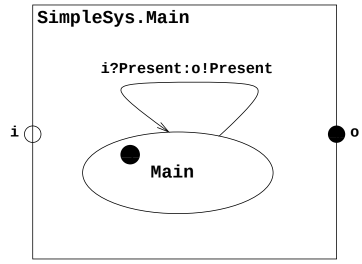


Figure 3.1: A Very Simple System

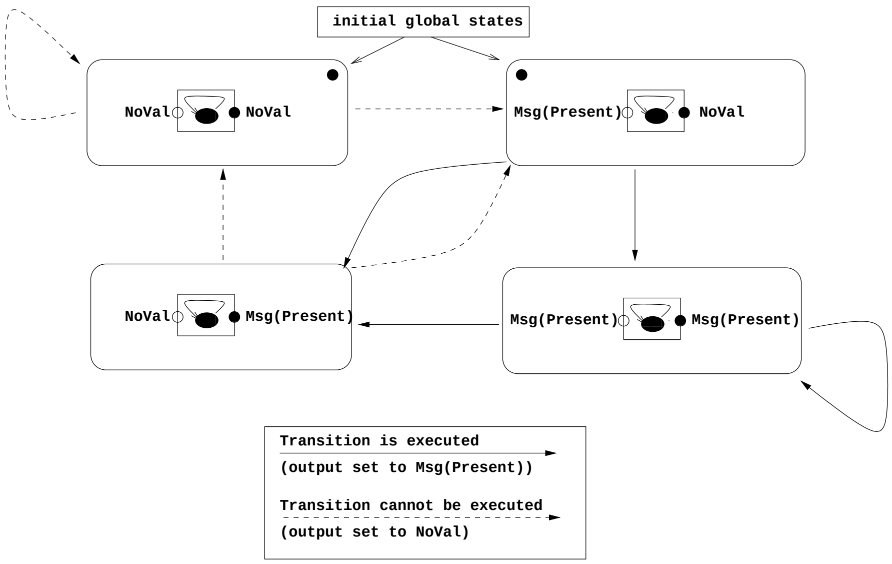


Figure 3.2: The Corresponding STG

3.1.2 Translation of an AUTOFOCUS Component Containing a STD

First, we show how to determine the state transition graph (i.e. state space, initial states and transition relation) for an AUTOFOCUS component with an extended finite automaton (STD) assigned to it that specifies its behaviour. Later on, we describe the translation of SSDs containing multiple components connected via channels (see section 3.1.3). We assume that the STDs are *flat*, i.e. no state contains a sub-STD. An informal description how general, hierarchical STDs can be converted into flat ones can be found in [PS99].

The State Variables

In each computation step, the global state of a component containing a STD is given by:

- the current control state of the automaton
- the values of the local variables of the component
- the data on the input and output ports

Together, these form the state variables of the corresponding state transition graph. For the control state, a new data type definition is added to the system specification, consisting of an enumeration of all possible states of the STD. For example, for the STD for the `Till` component in the cash point example given in chapter 2 (see Fig. 2.3), the new data type definition is

```
data Till.state_type = Ready | CardEntered | PINEntered |
                       Transmitting | WaitForAnswer ;
```

The data types for the local variables and ports are specified in the corresponding SSD. Note that the ports are of data type `Channel` and thus can also contain the empty message `NoVal`.

The Initial State

At the beginning of a computation, the STD is in its initial control state, all local variables are set to their initial value (unless no initial value is defined), and finally all output ports must contain the empty message `NoVal`. The values at the input ports are left open.

For example, the initial state of the `Till` component is given by the following term:

$$\begin{aligned} I(s) = \;& (\mathbf{state} == \mathbf{Ready}) \wedge (\mathbf{storedCard} == \mathbf{Invalid}) \wedge \\ & (\mathbf{transaction} == \mathbf{TA}(\mathbf{Withdraw}(0), 0)) \wedge (\mathbf{t} == 0) \wedge \\ & \mathbf{is\_NoVal}(\mathbf{out}) \wedge \mathbf{is\_NoVal}(\mathbf{Display}) \wedge \\ & \mathbf{is\_NoVal}(\mathbf{Eject}) \wedge \mathbf{is\_NoVal}(\mathbf{Money}) \end{aligned}$$

The Transition Relation

The transition relation for an AUTOFOCUS STD consists of the disjunction of one term T_{trans_i} for each transition in the STD and a term T_{idle} for the idle transition (that is executed if no transition is executable).

Remember that the translation is based on a synchronous semantics, so the idea is that each component with an STD assigned to it runs the program shown in Fig. 3.3 (see also [HS97]).

while True do
- select an applicable transition of the STD, depending on the values of the local variables and the messages on the input ports
- fire the transition: change the control state of the STD and the values of the local variables accordingly, and calculate the output data
- present the output data at the output ports, so it can be read in the next clock cycle
- wait for the next clock cycle

Figure 3.3: Behaviour of a Component Associated with a STD

Translation of a Single Transition

Therefore, the term T_{trans_i} corresponding to a single AUTOFOCUS transition $trans_i$ in an STD should evaluate to `True` if

- the values at the input ports match the input patterns of the transition
- the precondition is satisfied
- the output ports in the next state are set to the values specified in the output patterns
- the local variables in the next state are set to the values specified in the actions

.

This way, if $T_{trans_i}(s, s') = \mathbf{True}$, the transition $trans_i$ is executable in state s, and s' is a possible successor state. T_{trans_i} consists of a conjunction of subterms corresponding to input patterns, precondition, action and output patterns of the transition.

When translating an **input pattern**, mainly an expression has to be generated that determines whether a message has to be present at the corresponding port or not, and pattern matching has to be eliminated by using discriminator and selector functions (see section 2.4). Remember that for the ports there is a special data type `Channel`, defined as

```
data Channel(alpha) = NoVal | Msg(getVal:alpha);
```

The translation will be explained by a few examples:

input pattern	translation
`Keypad?`	$\mathbf{is\_NoVal}(\mathbf{Keypad})$
`Keypad?x`	$\mathbf{is\_Msg}(\mathbf{Keypad})$
`Keypad?` `Command(Withdraw(amt))`	$\mathbf{is\_Msg}(\mathbf{Keypad})\wedge$ $\mathbf{is\_Command}(\mathbf{getVal}(\mathbf{Keypad}))\wedge$ $\mathbf{is\_Withdraw}($ $\mathbf{CommandSel1}(\mathbf{getVal}(\mathbf{Keypad})))$

The last example shows how pattern matching is eliminated: the translation of an input pattern using discriminator and selector functions makes sure that the value at the input port is indeed of the required form, and all variables bound to the pattern are then replaced by terms consisting of the selector functions that correspond to the constructor functions in the pattern. So, if the variable `amt` occurs later on in the terms, it is replaced by

$\mathbf{WithdrawSel1}(\mathbf{CommandSel1}(\mathbf{getVal}(\mathbf{Keypad})))$.

(`WithdrawSel1` and `CommandSel1` are default selectors generated because no names for the selectors were given in the data type definition).

The translation of the **precondition** is then straightforward, and the **output patterns** and **actions** are just converted into equations containing the corresponding port names and local variables. Note that the execution of a transition affects the data values of the local variables and ports in the successor state, which is reflected by using the $\mathbf{next}$ operator. Besides, all output ports that are not mentioned in the output patterns are set to `NoVal`, and all local variables not mentioned in the action part keep their current values.

Finally, the term for a transition contains a subterm that makes sure that the current control state and the next control state are equal to the source and destination state of the transition.

As an example, the transition (9) in the component `Till` from state `PINEntered` to state `Ready` with the label:

```
amt>maxForToday(day,storedCard):
Keypad?Command(Withdraw(amt));RadioTime?day:
Display!ErrorDayLimitExceeded;Eject!storedCard:
storedCard = Invalid
```

is translated to the term

$$\begin{aligned}
&\mathbf{state} == \mathbf{PINEntered} \land \mathbf{next}(\mathbf{state}) == \mathbf{Ready} \land \\
&\\
&\mathbf{is_Msg}(\mathbf{Keypad})) \land \mathbf{is_Command}(\mathbf{getVal}(\mathbf{Keypad})) \land \\
&\mathbf{is_Withdraw}(\mathbf{CommandSel1}(\mathbf{getVal}(\mathbf{Keypad}))) \land \\
&\mathbf{is_Msg}(\mathbf{RadioTime}) \land \\
&\\
&\mathbf{WithdrawSel1}(\mathbf{CommandSel1}(\mathbf{getVal}(\mathbf{Keypad}))) > \mathbf{maxForToday}(\\
&\mathbf{getVal}(\mathbf{RadioTime}), \mathbf{storedCard}) \land \\
&\\
&\mathbf{getVal}(\mathbf{next}(\mathbf{Display})) == \mathbf{ErrorDayLimitExceeded} \land \\
&\mathbf{getVal}(\mathbf{next}(\mathbf{Eject})) == \mathbf{storedCard} \land \\
&\mathbf{is_NoVal}(\mathbf{next}(\mathbf{out})) \land \mathbf{is_NoVal}(\mathbf{next}(\mathbf{Money})) \land \\
&\\
&\mathbf{next}(\mathbf{storedCard}) == \mathbf{Invalid} \land \\
&\mathbf{next}(\mathbf{t}) == \mathbf{t} \land \mathbf{next}(\mathbf{transaction}) == \mathbf{transaction}
\end{aligned}$$

The Idle Transition

The idle transition T_{idle} specificies the behaviour of the system in case no transition of the STD is executable. The corresponding term consists of the negation of the disjunction of the preconditions and input pattern matching expressions for all transitions of the STD, and an expression stating that all output ports are set to `NoVal` and the local variables keep their current values.

The disjunction of the idle transition and the terms for all transitions of the STD finally yields the transition relation for the component.

3.1.3 Translation of a SSD Containing Multiple Components

In general, an AUTOFOCUS system consists of a hierarchical component network (see Fig. 3.4). This network is defined by a SSD and shows how the components are connected via channels. We assume that all basic components (the components that do not contain subcomponents) are assigned to a STD, so the behaviour of the system is completely specified. In addition, AUTOFOCUS allows only basic components to have local variables, so there can be no access conflicts concerning the variables. The execution of the system is then modelled as follows (based on the synchronous semantics):

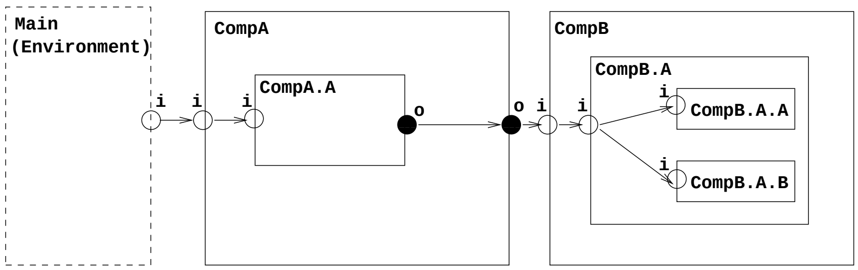


Figure 3.4: Hierarchical Component Network

All components run the program shown in section Fig. 3.3 in parallel and are synchronized by a global clock. The data presented at the output ports of a component is forwarded immediately via the chains of channels depicted in the SSD to be read by other components in the next clock cycle. Thus, the channels can be thought of as “electrical wires”. Output data is only buffered for one clock cycle at the output ports (some kind of "latches" when talking in hardware terms), thus allowing feedback of data into the same component.

We translate such a component network into a STG simply by combining the STGs for the basic components. The communication links are modelled by using a shared variable for all ports connected via a chain of channels. As a name for this shared variable, the port at the beginning of the chain is used. Wherever the name of another port belonging to the chain occurs in the terms for the basic components, it is substituted by the name of the port at the beginning of the chain.

As an example, in Fig. 3.4, all occurrences of port `i` in the basic components `CompB.A.A` and `CompB.A.B` will be substituted by port `o` in component `CompA.A`, and all occurrences of port `i` in component `CompA.A` will be replaced by port `i` in the environment `Main`.

Now, with the above mentioned substitution of port names applied to the state variables and the terms for the initial state and the transition relation of the basic components, the translation of a component network into a STG can be given as follows:

- The **state space** consists of the union of the state variables of all basic components.
- The term $I(s)$ for the **initial state** is the conjunction of the terms for the initial states of the basic components.
- The term $T(s, s')$ for the **transition relation** is the conjunction of the transition relations of the basic components.

3.1.4 Specifying Execution Sequences

With the help of the translation of AUTOFOCUS system models into state transition graphs we can now describe possible executions of a system up to some point in time. Suppose we are interested in the system behaviour during the first $k + 1$ steps. Then we have to consider $k + 1$ copies of the state variables: at the beginning, the system is in the global state s_0 (characterized by particular values of the input and output ports, control states of the automata and local variables). At the first clock tick, transitions are chosen in the automata, depending on the values of the state variables in state s_0, and executed, and other data values appear at the input ports. This puts the system into state s_1, etc.

We call the sequence $s_0, s_1, ..., s_k$ an *execution sequence*. For each step, the execution sequence specifies particular values for the state variables (the data on the input and output ports, the control states of the automata and the values of the local variables). We denote the state variables of state s_i (i.e., the i-th step of the execution sequence) as $v_1.i, \ldots, v_n.i$.

Every execution sequence corresponds to a path of length k in the state transition graph, starting at an initial state. Thus, we can characterize an arbitrary execution sequence of an AUTOFOCUS system we have translated into a state transition graph as follows (see Fig. 3.5): s_0 must be the initial state, so $I(s_0)$ must be true. s_1 can be the global state in the next execution step, if (s_0, s_1) is in the transition relation, i.e. if $T(s_0, s_1)$ is true, etc. If we perform this "unfolding of the transition relation" k times, we have

$M = I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \ldots T(s_{k-1}, s_k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_i + 1)$.

The boolean term M now describes **all possible execution sequences** with length $k+1$ of the system, because it evaluates to `True` for all sequences of global states that are valid execution sequences of the system we specified. If we add other predicates to M, we can describe specific execution sequences we are interested in (for example, we want a certain component to be in a certain state in some step).

To determine a particular execution sequence, we now have to find values for the state variables $v_j.i$ at the different steps, so that M evaluates to `True`. However, I and T still contain the complex AUTOFOCUS data types and functions. In the following sections, we show how to eliminate these and to transform M into a formula in propositional logic.

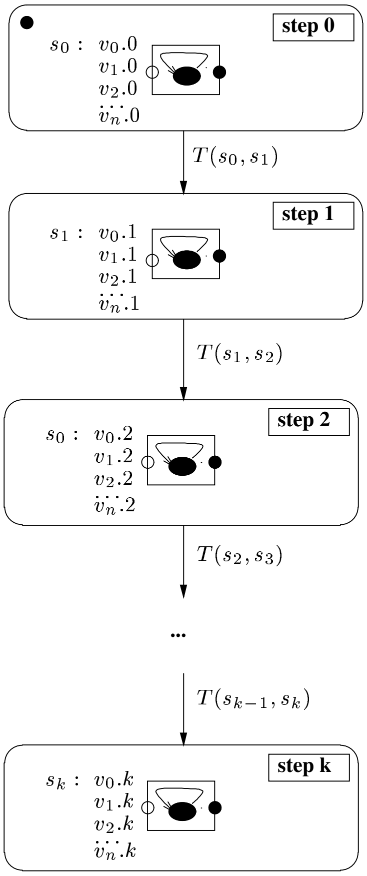


Figure 3.5: Determining Execution Sequences

Then, we will explain how to calculate a solution to such a propositional formula with the help of the satisfiability solver SATO.

As an example, Fig. 3.6 shows a 4-step execution sequence for the simple system in Fig. 3.1. In this case, the only transition in the system is executed three times in a row, so in step 0, 1 and 2 the value at port `i` must be `Msg(Present)`, and because of the execution of the transition, the value at port `o` in the following step (1, 2 and 3) must also be `Msg(Present)`. The reader can make sure himself that this execution sequence really corresponds to a path in the state transition graph given in Fig. 3.2.

3.2 Eliminating User-Defined Data Types and Functions

In this section, we show how to eliminate user-defined functions and data types (as defined in section 2.4) from an AUTOFOCUS system description. This is achieved by substituting all function applications by their definition and by mapping the values of the user-defined data types to integer numbers. All constructor, selector and discriminator terms are then replaced accordingly to manipulate those integer values. The elimination of user-defined data types and functions was kept strictly separate from the other translation steps described in this thesis, and was also implemented in a separate JAVA class for the following reasons:

- Most simulation and verification tools and programming languages do not support hierarchical Gofer-like data types. Therefore, a system description using only low-level data types (as integers) is generally needed to integrate such tools into AUTO-

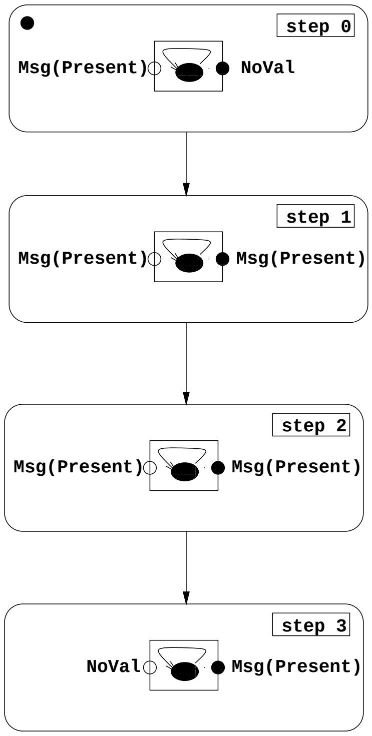


Figure 3.6: Execution Sequence for Simple System

FOCUS. The concept described in this chapter can also be used to facilitate the translation of AUTOFOCUS system models into the SMV input language, the SPIN input language, a JAVA simulation program, etc.

- It is possible to define different mappings of hierarchical data types to integer values. With separate code for the mapping, it is very easy to implement various methods and to switch between them. In this section, we will show two mappings and compare their efficiency.

The concept of eliminating user-defined AUTOFOCUS data types and functions has already been employed in the SMV translation [PS99] and thus will be explained fairly briefly. However, we also present an alternative mapping of hierarchical data types to integer values to the one described in [PS99], which leads to a much more efficient representation.

3.2.1 Eliminating User-Defined Functions

Elimination of user-defined functions in AUTOFOCUS system models is done exactly as shown in [PS99], by replacing them by their definition. If a function definition consists of more than one alternative and uses pattern matching, a `Case` term is generated and the pattern matching expressions are eliminated in the same way as shown in section 3.1.2 when translating input patterns using discriminator and selector terms. Finally, `Const` declarations are treated as functions without parameters, their elimination is straightforward.

Note that for this method to be applicable, functions must not be defined recursively.

Example

As an example, we will show the replacement term for the function `criticalAction` given in section 2.4:

function definition

```
fun
    criticalAction(Withdraw(x)) = x>500
  | criticalAction(SendMail) = False
  | criticalAction(ViewBalance) = False;
```

replacement

$$\begin{array}{ll}\mathbf{case} & \\ \quad \mathbf{is\_Withdraw(X0)}: & \mathbf{WithdrawSel1(X0)} > 500 \\ \quad \mathbf{is\_SendMail(X0)}: & \mathbf{False} \\ \quad \mathbf{is\_ViewBalance(X0)}: & \mathbf{False} \\ \mathbf{esac} & \end{array}$$

Every occurrence of the function `criticalAction` will then be replaced by the term shown above, e.g. `criticalAction(act)` in a precondition will become
$\mathbf{case\ is\_Withdraw(act)}: \mathbf{WithdrawSel1(act)} > 500 \ldots \mathbf{esac}$
If necessary (i.e. if the definition of a function contains other functions), the replacement is carried out recursively.

3.2.2 Eliminating User-Defined Data Types

Now, we show how to eliminate AUTOFOCUS data types by mapping them to integer values. However, we restrict ourselves to *finite* types, because most model checking and verification tools (including our method to determine execution sequences with SATO) assume that the state space is finite. This leaves us with the following two possible data types:

- **`Int`**: this is the AUTOFOCUS (unsigned) integer data type, it will just be mapped to itself. We only have to take care that a maximum value `maxInt` for this data type is defined, as can be done within the QUEST framework.
- **User defined data types** (see section 2.4), of the following general form:

$$\begin{array}{rl} data\ Type = & C_1(sel_{11} : Type_{11}, \ldots, sel_{1n_1} : Type_{1n_1}) | \ldots | \\ & C_m(sel_{m1} : Type_{m1}, \ldots, sel_{mn_m} : Type_{mn_m}) \end{array}$$

where no data type definition must be based directly or indirectly on itself. The n_i can also be 0, meaning that the alternative C_i has no arguments. For these data type definitions, we have to give a mapping to integer values, and we have to show how to translate selector, constructor and discriminator expressions into terms manipulating these integer values.

3.2.3 Encoding by Enumeration

This is the method of encoding user-defined types to integers that was used in the translation to SMV [PS99]. The main idea of this encoding is that all possible values for a data type are enumerated and consecutively mapped to integer numbers starting from 0. The possible values for a data type are calculated by determining all the values that are generated by the constructors $C_1, \ldots, C_m$, as follows:

- If the constructor C_i has no arguments, C_i itself is the only value generated using this constructor.

- If C_i has arguments, apply C_i to all possible combinations of the values of the arguments (which are calculated recursively).

Now, the constructor, selector and discriminator functions defined for each data type have to be translated to functions operating on the corresponding integer values:

- The **constructor function** $C_i(\ldots)$ for the i-th alternative is translated to a case distinction. For each combination of the integer values for the arguments $(s_1, \ldots, s_m)$, it gives the integer value for $C_i(s_1, \ldots, s_m)$ as a result.
- The **selector function** `sel`$_j$`(x)` for the j-th argument of alternative C_i is also translated to a case distinction. For each integer value mapped to one of the terms generated by the constructor C_i, it gives the integer value of the j-th argument s_j of C_i as a result.
- The **discriminator function** `is_`$C_i(x)$ is translated to a disjunction that evaluates to `True` if x is one of the integer values mapped to a term generated by C_i.

Example

How this encoding works can best be shown by an example. We take a variant of the data type `Card` (see section 2.4), with the card date left out:

```
data Cardinfo = CI(PINCode:Int, accNo:Int);
data Card     = Invalid | Valid(info:Cardinfo, dayAmount: Int);
```

Fig. 3.7 shows the enumeration encoding of this type, assuming a maximum integer value of 1. The translation of the corresponding constructor, discriminator and selector terms is given in Fig. 3.8.

Type Cardinfo		**Type Card**	
CI(0,0)	0	Invalid	0
CI(1,0)	1	Valid(CI(0,0),0)	1
CI(0,1)	2	Valid(CI(0,1),0)	2
CI(1,1)	3	Valid(CI(1,0),0)	3
		Valid(CI(1,1),0)	4
		Valid(CI(0,0),1)	5
		Valid(CI(0,1),1)	6
		Valid(CI(1,0),1)	7
		Valid(CI(1,1),1)	8

Figure 3.7: Enumeration Encoding for Type `Card`

The Channel Data Type

As stated in section 2.4, channels are defined as

```
data Channel(alpha) = NoVal | Msg(getVal: alpha) ;
```

In principle, we could translate this special data type as described above. However for efficiency reasons an optimisation is used: `Msg(X)` is mapped to itself, and `NoVal` is mapped to the maximum integer value generated for the data type `alpha`, plus 1.

This way, the translation of the data type `Channel(Card)` is as depicted in Fig. 3.9.

constructors		**discriminators**	
`Invalid:`	0	`is_Invalid(X):`	$\mathbf{X} == 0$

`Valid(X0,X1):`	`case`
	$(\mathbf{X0} == 0) \wedge (\mathbf{X1} == 0) : 1$
	$(\mathbf{X0} == 1) \wedge (\mathbf{X1} == 0) : 2$
	$(\mathbf{X0} == 2) \wedge (\mathbf{X1} == 0) : 3$
	$(\mathbf{X0} == 3) \wedge (\mathbf{X1} == 0) : 4$
	$(\mathbf{X0} == 0) \wedge (\mathbf{X1} == 1) : 5$
	$(\mathbf{X0} == 1) \wedge (\mathbf{X1} == 1) : 6$
	$(\mathbf{X0} == 2) \wedge (\mathbf{X1} == 1) : 7$
	$(\mathbf{X0} == 3) \wedge (\mathbf{X1} == 1) : 8$
	`esac`
		`is_Valid(X):`	$(\mathbf{X} == 1)\vee$
			$(\mathbf{X} == 2)\vee$
			$(\mathbf{X} == 3)\vee$
			$(\mathbf{X} == 4)\vee$
			$(\mathbf{X} == 5)\vee$
			$(\mathbf{X} == 6)\vee$
			$(\mathbf{X} == 7)\vee$
			$(\mathbf{X} == 8)$

selectors			

`info(X):`	`case`
	$(\mathbf{X} == 1) : 0$
	$(\mathbf{X} == 2) : 1$
	$(\mathbf{X} == 3) : 2$
	$(\mathbf{X} == 4) : 3$
	$(\mathbf{X} == 5) : 0$
	$(\mathbf{X} == 6) : 1$
	$(\mathbf{X} == 7) : 2$
	$(\mathbf{X} == 8) : 3$
	`esac`
		`dayAmount(X):`	`case`
			$(\mathbf{X} == 1) : 0$
			$(\mathbf{X} == 2) : 1$
			$(\mathbf{X} == 3) : 0$
			$(\mathbf{X} == 4) : 1$
			$(\mathbf{X} == 5) : 0$
			$(\mathbf{X} == 6) : 1$
			$(\mathbf{X} == 7) : 0$
			$(\mathbf{X} == 8) : 1$
			`esac`

Figure 3.8: Functions for Type `Card` (Enumeration Encoding)

constructors		**discriminators**	
`NoVal:`	9	`is_NoVal(X):`	$\mathbf{X} == 9$
`Msg(X):`	$\mathbf{X}$	`is_Msg(X):`	$\mathbf{not}(\mathbf{X} == 9)$
selector			
`getVal(X):`	$\mathbf{X}$		

Figure 3.9: Translation of Type `Channel(Card)` (Enumeration Encoding)

Drawbacks of the Encoding by Enumeration

As can be seen from the example, the enumeration encoding can lead to very large replacement terms for the selectors, constructors and discriminators (imagine we had used a maximum integer value of 15 instead of 1!). This makes the resulting transition relation of the AUTOFOCUS system very complicated and difficult to be used as input for verification or determining execution sequences.

One could achieve shorter terms by using multiplication, division and modulo operators instead of the case distinctions. However, we later have to represent the variables as bit sequences and translate the operators accordingly (see section 3.3). For multiplication, division and modulo (if not by a power of two), this would again lead to very complex terms.

3.2.4 Encoding Using Bitfields

Because of the problems of the enumeration encoding in [PS99], we have developed an alternative solution to map AUTOFOCUS data types to integer numbers: the values of the data type

$$data\,Type = \begin{array}{l} C_1(sel_{11} : Type_{11}, \ldots, sel_{1n_1} : Type_{1n_1}) | \ldots | \\ C_m(sel_{m1} : Type_{m1}, \ldots, sel_{mn_m} : Type_{mn_m}) \end{array}$$

are represented by bitfields (see Fig. 3.10). The lower bits always contain the "tag", a number from 0 to $(m-1)$ corresponding to the terms generated by C_1, C_2, ..., C_m. If the constructor C_i has arguments, their integer representations (which are bitfields as well) are prefixed to the tag. As the bitfields for the different alternatives can have different lengths, unneeded space is filled with 0.

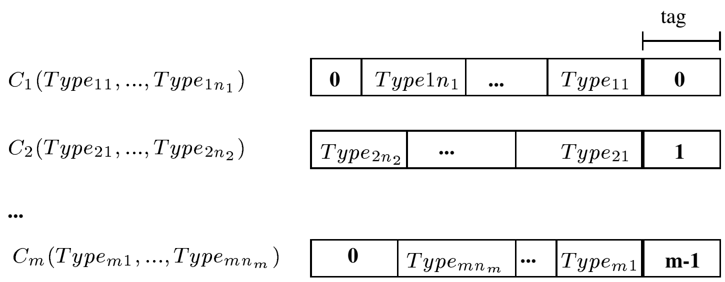


Figure 3.10: Bitfield Encoding

The translation of constructor, selector and discriminator terms uses operations on bitfields and is quite straightforward:

- The **constructor function** $C_i(s_1, \ldots, s_m)$ for the i-th alternative creates a bitfield containing $i-1$ as the tag and the integer values for the arguments prefixed to it.
- The **selector function** `sel`$_{ij}$`(x)` for the j-th argument of alternative C_i extracts the integer value for that argument from the bitfield.
- The **discriminator function** `is_`$C_i(x)$ extracts the tag from the bitfield and compares it to $(i-1)$, the tag number of the alternative C_i.

Example

Now, we will show the encoding and the constructor, discriminator and selector terms for the example shown in the last section. Fig. 3.11 gives the bitfield representation for the data types and Fig. 3.12 shows the corresponding functions.

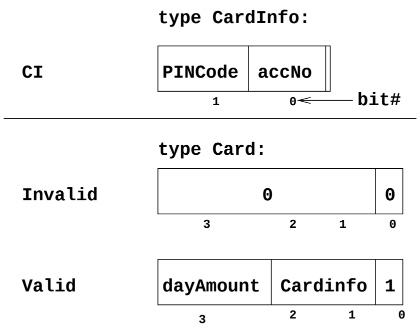


Figure 3.11: Bitfield Encoding for Type `Card`

Note that the data type Cardinfo has only one constructor, so there is no need to reserve space for the tag.

The actual operations on bitfields are done via the operators $+$, $\cdot$, div and mod because these operators are also included in the SMV input language. However, the operand is always a power of two, so the actual implementation of the operators can be based on bit shift and extract operations.

constructors	**discriminators**
`Invalid:` 0	`is_Invalid(X):` $(\mathbf{X} \mathbf{\ mod\ } 2) == 0$
`Valid(X0,X1):` $0 + \mathbf{X0} \cdot 2 + \mathbf{X1} \cdot 8$	`is_Valid(X):` $(\mathbf{X} \mathbf{\ mod\ } 2) == 1$
	selectors
`info(X):` $(\mathbf{X} \mathbf{\ div\ } 2) \mathbf{\ mod\ } 4$	`dayAmount(X):` $(\mathbf{X} \mathbf{\ div\ } 8) \mathbf{\ mod\ } 2$

Figure 3.12: Functions for Type Card (Bitfield Encoding)

The Channel Data Type

The data type Channel is treated in the same way as shown in section 3.2.3, by mapping NoVal to an integer value not used in the bitfield representation. We also take the maximum possible value for the bitfield, plus 1.

Illegal values

In this section, we have given an integer representation of the AUTOFOCUS data types that leads to short terms for the constructors, selectors and discriminators. Moreover, these terms can be converted very easily to terms operating on single bits of the state variables — we only need bit shift and extract operations.

Unfortunately, this solution also contains a serious problem: the mapping is not consecutive anymore ! There are integer numbers between 0 and the maximum possible integer value for the data type that cannot be generated by any constructor terms. For example, 2 is not a possible integer value for the data type Card.

As the execution sequence determination cannot distinguish between legal and illegal values (it does not see the underlying hierarchical data types anymore), it might calculate impossible input data.

To solve this problem, we give a condition that takes the integer value for a data type as an argument and makes sure it is legal. We then have to add these conditions to the formulas for the initial states and the transition relation to avoid illegal values on input ports (there is no need to specify that the values of the output ports and local variables have to be legal, as the transitions always calculate only legal values).

The condition ensuring legal values for a data type consists of one subterm that specifies that the tag is less or equal than its maximum value, and a subterm for each alternative that specifies that the values for all selectors have to be legal as well and the rest of the bitfield has to be filled with 0.

As an example, we show the condition for legal values for the data type Card:

```
data Card = Invalid | Valid(info: Cardinfo, dayAmount: Int);
```

Component	max. Int	bitfield enc. #clauses	time[s]	enumeration enc. #clauses	time[s]
Till	1	1452	16.6	3441	287.8
	2	2317	18.9	—	>.5h
	3	2391	21.6	—	—
	4	3232	27.9	—	—
Connection	1	222	3.0	222	3.4
	2	276	2.9	222	4.3
	3	276	2.7	222	9.1
	4	330	2.9	222	23.7
Driver	1	317	2.5	319	3.2
	2	416	2.5	319	4.3
	3	416	2.6	336	9.3
	4	515	2.9	372	23.7
Process	1	63	8.4	300	6.9
	2	77	8.2	885	41.1
	3	77	9.4	1651	116.1
	4	91	8.5	3133	786.3

Figure 3.13: Comparison of the Encodings

$$
\begin{aligned}
&\mathbf{legal\_value\_Card}(\mathbf{X}) = \\
&\quad ((\mathbf{X}\ \mathbf{mod}\ 2) \leq 1) \wedge \\
&\quad (\mathbf{is\_Invalid}(\mathbf{X}) \Rightarrow ((\mathbf{X}\ \mathbf{div}\ 2) == 0)) \wedge \\
&\quad (\mathbf{is\_Valid}(\mathbf{X}) \Rightarrow \\
&\qquad (\mathbf{legal\_value\_Cardinfo}(\mathbf{info}(\mathbf{X})) \wedge \mathbf{legal\_value\_Int}(\mathbf{dayAmount}(\mathbf{X})))
\end{aligned}
$$

For the data type `Channel<alpha>`, `NoVal` is also a legal value.

3.2.5 Comparison of the Two Encodings

The bitfield encoding leads to much shorter terms for the constructor, discriminator and selector functions. Should it therefore be the preferred method to translate hierarchical data types to integer terms ?

To answer this question, we measured the size of the transition relation for different components of the banking system using both encodings. The results can be seen in table 3.13. The size of the transition relation is given as the number of clauses of its conjunctive normal form (CNF). We will show later how the conjunctive normal form is generated (see section 3.3), for now it is sufficient to know that the number of clauses directly determines the time needed to calculate execution sequences with SATO. Table 3.13 also shows the execution time[1] needed to generate the CNF.

We can see that the bitfield encoding is by far superior to the enumeration encoding when complex terms using hierarchical data types are contained in the specification (as in `Till` and `Process`). The transition relation for `Till` was built in less than half a minute for maximum integer values up to 4 using the bitfield encoding. With the enumeration encoding, we had to abort the calculation after half an hour even with a maximum integer value of only 2.

The reason why the enumeration encoding is slightly better than the bitfield encoding for the components `Connection` and `Driver` is that the State Transition Diagram of these components does not contain any constructor, selector or discriminator terms at all.

[1] All execution times were measured on a SUN UltraSparc 2 with 1 GB main memory

So, the size of the transition relation is only determined by the number of bits used by the data types. This number is higher for the bitfield method because not all possible values are used. The execution time is still longer for the enumeration encoding, but only because the (very big) constructor, selector and discriminator terms are always pre-calculated in our implementation.

For the reasons given above, we consider the bitfield encoding to be by far preferable to the enumeration encoding. **All performance figures shown later in this thesis for specification based test sequence determination will be based on the bitfield encoding.**

3.2.6 Translation of AUTOFOCUS Systems to SMV

We have achieved an important intermediate result at this point: we can now translate an AUTOFOCUS system description into the SMV input language [McM93]. In the SMV input language, a system can be specified exactly as we have done it in section 3.1: by giving a state transition graph using boolean valued terms for the initial states (in SMV, this is done by the `INIT` statement), and for the transition relation (in SMV: `TRANS`). After the elimination of user-defined data types and functions from these terms, they are only based on integer values — a data type that SMV supports.

This gives us the possibility to verify properties of a specification using the model checker SMV, as an alternative to the SMV translation described in [PS99].

3.3 Boolean Encoding of the State Space

With the concepts explained in section 3.2, we can now eliminate the user-defined data types and functions from the terms $I(s)$ for the initial states and $T(s, s')$ for the transition relation. As a result, we only have expressions based on one type, integer values ranging from 0 up to some maximum value.

However, to be able use SATO to determine solutions for the formula M describing possible execution sequences (see section 3.1.4), we have to translate $I(s)$ and $T(s, s')$ to *propositional* formulas. That means, they must only contain boolean valued variables and standard boolean operators. For this reason, we need a state space consisting only of boolean variables.

In this section, we explain how to encode the state space using boolean variables and how to translate $I(s)$ and $T(s, s')$ accordingly. The translations of I and T will be expressions containing only the boolean state variables and the standard boolean operators $\mathbf{not}$, $\vee$, $\wedge$, $\Rightarrow$ and $\Leftrightarrow$.

3.3.1 Representing the State Space by Boolean Variables

Representing the state space by boolean variables is a very easy task: we just separate the integer valued state variables into their bits. Thus, for each state variable v_i we define

- The width $|v_i|$, the number of bits needed by the maximum integer value of v_i, and
- $|v_i|$ new state variables, each one referring to one bit of v_i. We will denote these new variables by $v_i@0, \ldots, v_i@(|v_i| - 1)$, where $v_i@0$ is the least significant bit of v_i etc.

3.3.2 Translation of the Terms for I and T

Translating I and T so that they are defined on the single bits of the state variables instead of their integer values is a bit more complicated. As an example, for the term

$\mathbf{timeout} == 4$

the result of the translation should be

$(\mathbf{timeout}@2 \Leftrightarrow 1) \wedge (\mathbf{timeout}@1 \Leftrightarrow 0) \wedge (\mathbf{timeout}@0 \Leftrightarrow 0)$

(As 4 is 100 in binary notation, the bits of $\mathbf{timeout}$ must be equal to 1, 0 and 0).

Before we describe how to translate I and T so that they are defined on the bits of the state variables, we first have to clarify which kinds of terms we have to consider. The terms for I and T resulting from the translation of an AUTOFOCUS system model and the elimination of AUTOFOCUS data types and functions as explained in sections 3.1 and 3.2 are of the following form:

```
Term ::=  StateVar | next(StateVar) | IntConst |
          not (Term) | Term Op Term |
          Term * IntConst | Term mod IntConst |
          Term div IntConst |
          If Term Then Term Else Term |
          Case Term:  Term; Term:  Term; ...; Esac;
Op ::=    == | ∨ | ∧ | ⇒ | ⇔ | > | ≥ | < | ≤ | + | - ;
```

We use a similar semantics for terms as in the programming language C: the only data type are integer numbers. We identify 0 with $\mathbf{False}$ and integer numbers different from 0 with $\mathbf{True}$. Operators with a boolean value as a result (e.g. $\mathbf{not}$, $==$, $\wedge$, $>$ etc.) return 0 for $\mathbf{False}$ and 1 for $\mathbf{True}$.

The idea of our translation is that we separate the value of a general term t into its bits. We use the same notation as for the state variables:

- $|t|$ is the maximum number of bits needed by the integer value of term t, and
- $t@i$ stands for the i-th bit of the integer value of this term.

Then we show how to calculate the bits $t@i$ and the width $|t|$ for a composed term like "$t_1 + t_2$" using the bits of its subterms. This is very similar to the way such operations are implemented in hardware circuits like the arithmetic-logic unit (ALU) in a processor: the operands are represented by bit sequences and the ALU contains a network of AND, OR and NOT gates that determines the bits of the result. The interested reader is referred to literature about computer architecture (e.g. [Bar91]) for details.

If we apply this algorithm recursively (i.e. showing how to calculate the bits of t using the bits of the subterms of t, which are again calculated using the bits of their subterms etc.), the last stage will be the bits of the state variables themselves (`StateVar@i`, and in the transition relation also `next(StateVar)@i` to refer to the successor state), or the bits of integer constants (like "5"). Thus, the result is a (potentially quite large) expression for each bit of t that only contains the bits of the state variables, the constants 0 and 1 (the bits of the integer constants) and boolean operators.

We consider t to be true, if its value is different from zero, i.e. one of its bits is 1:
$\mathbf{is\_True}(t) = \bigvee_{i=0}^{|t|-1} t@i$

Therefore, if we apply this approach to the terms I and T for the initial states and the transition relation, then

- $\mathbf{is\_True}(\mathbf{I})$ and
- $\mathbf{is\_True}(\mathbf{T})$

are the desired expressions for I and T, containing only the bits of the state variables, the constants 0 and 1 and the boolean operators $\mathbf{not}$, $\vee$, $\wedge$, $\Rightarrow$ and $\Leftrightarrow$.

Now, the only thing we still have to do is to show how to calculate the bits $t@i$ and the width $|t|$ for the different kinds of terms. This will be explained in the following sections.

Boolean Operators

Terms with boolean operators ($t = \mathbf{not}(a)$, $t = a \wedge b$, $t = a \vee b$, $t = a \Rightarrow b$, $t = a \Leftrightarrow$ b) return 1 for $\mathbf{True}$ and 0 for $\mathbf{False}$. So for these terms, $|t| = 1$ (the value of t fits into one bit), and:

- $(\mathbf{not}(a))@0 = \mathbf{not}(\mathbf{is\_True}(a))$
- $(a \circ b)@0 = \mathbf{is\_True}(a) \circ \mathbf{is\_True}(b)$, for $\circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$

Comparison Operators

The comparison operators $==$, $>$, $\geq$, $<$, $\leq$ also return 1 resp. 0 for **True** or **False**. a is equal to b if all bits of a and b are equal. a is greater than b if the highest bit of a is 1 and the highest bit of b is 0, or if their highest bits are equal and the second highest bit of a is 1 and the second highest bit of b is 0 etc. So we can determine $a == b$ and $a > b$ from the bits of a and b as follows ($n = \max(|a|, |b|)$ denotes the maximum number of bits needed by a and b):

- $|a == b| = |a > b| = 1$ (the result of $a == b$ and $a > b$ is either 0 or 1, which takes 1 bit)
- $(a == b)@0 = \bigwedge_{i=0}^{n-1} (a@i \Leftrightarrow b@i)$
- $$\begin{aligned}(a > b)@0 = \; & (a@(n-1) \wedge \mathbf{not}(b@(n-1)))\vee \\ & ((a@(n-1) \Leftrightarrow b@(n-1))) \wedge (a@(n-2) \wedge \mathbf{not}(b@(n-2))))\vee \\ & \ldots \vee \\ & ((a@(n-1) \Leftrightarrow b@(n-1)) \wedge \ldots \wedge \\ & \qquad (a@1 \Leftrightarrow b@1)) \wedge (a@0 \wedge \mathbf{not}(b@0)))\end{aligned}$$

The term $a \geq b$ is treated as $(a > b) \vee (a == b)$, and $a < b$ and $a \leq b$ are treated as $b > a$ and $b \geq a$ respectively.

If-Terms

The term
$t =$ `If` **Condition** `Then` **ThenTerm** `Else` **ElseTerm**
represents the value of **ThenTerm** if **Condition** is true, and the value of **ElseTerm** otherwise. The maximum number of bits taken by this term is either the maximum number of bits taken by **ThenTerm** or the maximum number of bits taken by **ElseTerm**, whichever is greater. Thus:

- $|t| = \max(|\mathbf{ThenTerm}|, |\mathbf{ElseTerm}|)$
- $$\begin{aligned}t@i = \; & \mathbf{is\_True}(\mathbf{Condition}) \Rightarrow \mathbf{ThenTerm}@i \wedge \\ & \mathbf{not}(\mathbf{is\_True}(\mathbf{Condition})) \Rightarrow \mathbf{ElseTerm}@i\end{aligned}$$

Case-Terms

Case terms are a generalisation of If-Terms:

$t =$ `Case`
  $\mathbf{Condition}_1$: $\mathbf{Result}_1$
  $\mathbf{Condition}_2$: $\mathbf{Result}_2$
  ...
  $\mathbf{Condition}_n$: $\mathbf{Result}_n$
`Esac`

If $\mathbf{Condition}_1$ is true, the value of the case term is the value of $\mathbf{Result}_1$, otherwise if $\mathbf{Condition}_2$ is true, the value of the case term is the value of $\mathbf{Result}_2$ etc.:

- $|t| = \max_i |\mathbf{Condition}_i|$

- $t@i = \quad (\textbf{is\_True}(\textbf{Condition}_1) \Rightarrow \textbf{Result}_1@i) \wedge$
  $((\textbf{not}(\textbf{is\_True}(\textbf{Condition}_1)) \wedge \textbf{is\_True}(\textbf{Condition}_2)) \Rightarrow$
  $\textbf{Result}_2@i) \wedge \ldots \wedge$
  $((\textbf{not}(\textbf{is\_True}(\textbf{Condition}_1) \vee \textbf{is\_True}(\textbf{Condition}_2) \vee \ldots \vee$
  $\textbf{is\_True}(\textbf{Condition}_{n-1})) \wedge \textbf{is\_True}(\textbf{Condition}_n)) \Rightarrow$
  $\textbf{Result}_n@i)$

Multiplication Operators

In our current implementation, the multiplication operators $\cdot$, $\mathbf{div}$, $\mathbf{mod}$ can only be translated if the right operand is a power of two (i.e., an integer constant of the form 2^j). In this case, we can use simple bit shift and extraction operations:

- $|a \cdot 2^j| = |a| + j$
- $(a \cdot 2^j)@i = \begin{cases} a@(i-j) & \text{if } i \geq j \\ 0 & \text{otherwise} \end{cases}$
- $|a \;\mathbf{div}\; 2^j| = |a| - j$
- $(a \;\mathbf{div}\; 2^j)@i = a@(i+j)$
- $|a \;\mathbf{mod}\; 2^j| = j$
- $(a \;\mathbf{mod}\; 2^j)@i = \begin{cases} a@i & \text{if } i < j \\ 0 & \text{otherwise} \end{cases}$

This is all we need if we assume that multiplication operators do not occur in the terms for preconditions, output patterns and actions of the AUTOFOCUS transitions — then the only multiplication operators are introduced by the translation of the data types using the bitfield encoding (see section 3.2.4). In this case, the right operand is always a power of two, as the multiplication operands were only used as a notation for operations on the bitfields (like extracting a value).

A general translation of terms with multiplication operators into terms depending on the bits of the operands would be very complex (the corresponding hardware circuit is called "parallel multiplier", see [Bar91]), we leave its implementation as further work.

Addition

To understand how addition is treated, consider Fig. 3.14. It shows how a and b are added bit by bit, starting from bit 0.

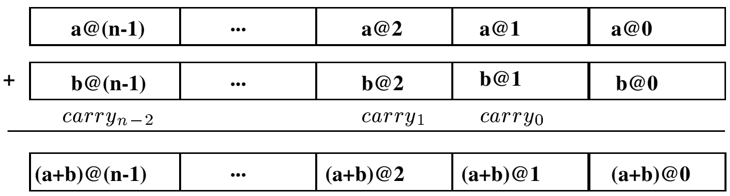


Figure 3.14: Addition

To determine the bits of $t = a + b$, we use the operator $\oplus$ (exclusive or, resp. addition modulo two, with $0 \oplus 0 = 0$, $0 \oplus 1 = 1 \oplus 0 = 1$, $1 \oplus 1 = 0$). $x \oplus y$ is the same as $\mathbf{not}(a \Leftrightarrow b)$, so we actually do not need to introduce a new operator.

If we add the bits of a and b as shown in Fig. 3.14, we have

- $(a + b)@0 = a@0 \oplus b@0$
- $(a + b)@i = a@i \oplus b@i \oplus carry_{i-1}$ for $0 < i < n$

The carry bit $carry_i$ is 1 if at least two of the bits in the corresponding column are 1:

- $carry_0 = a@0 \land b@0$
- $carry_i = (a@i \land b@i) \lor (a@i \land carry_{i-1}) \lor (b@i \land carry_{i-1})$ for $0 < i < n$

The method described above is exactly the way the binary addition is usually implemented in hardware, as a "parallel adder circuit" [Bar91].

Subtraction

The subtraction $a - b$ can also be seen as an addition:
$a - b = a + (-b)$.

$-b$ is the two's complement of b and can be determined by negating all its bits (denoted as $\overline{b}$) and adding 1.

Thus, we replace $a - b$ by $a + \overline{b} + 1$ and use the method described above (actually, the "+1" is treated like a special carry bit $carry_{-1}$ that is automatically added to $a@0$ and $b@0$, and the expressions for the bits of $a + b$ are modified accordingly). See [Bar91] for a detailed description of the two's complement notation and bitwise subtraction of binary numbers.

The width $n = |a + b|$ resp. $n = |a - b|$ of the result will be fixed to the width of the AutoFocus `Integer` type, to avoid problems with adding negative numbers (both arguments must have the same number of bits) or overflow (if the result has more bits than the arguments).

Increment and Decrement

The frequently used terms $a+1$ and $a-1$ are treated in a special way, to avoid unnecessarily complicated results.

Increment

From the example in Fig. 3.15 it can easily be seen that $carry_i$ is 1 as long as all bits of a from bit 0 to bit i are 1:

	0	1	0	1	1	1	1	(47)
+	0	0	0	0	0	0	1	(1)
	0	0	1	1	1	1		
	0	1	1	0	0	0	0	(48)

Figure 3.15: Increment Example

$$carry_i = \begin{cases} 1 & \text{if} a@0 \land a@1 \ldots \land a@i \\ 0 & \text{otherwise} \end{cases}$$

Thus,

- $(a + 1)@0 = a@0 \oplus 1 = \mathbf{not}(a@0)$
- $(a + 1)@i = \quad a@i \oplus 0 \oplus carry_{i-1} =$
  $a@i \oplus (a@0 \land a@1 \land \ldots \land a@(i-1))$ for $0 < i < n$

Decrement

The same idea can be used for $a-1$ (see Fig. 3.16). The two's complement of 1 (representing -1) is binary 1....1, so $carry_i$ is 0 if all bits from bit 0 to bit i are 0:

	0	1	1	0	0	0	0	(48)
+	1	1	1	1	1	1	1	(-1)
	1	1	0	0	0	0		
	0	1	0	1	1	1	1	(47)

Figure 3.16: Decrement Example

$$carry_i = \begin{cases} 0 & \text{if } \mathbf{not}(a@0) \wedge \mathbf{not}(a@1) \ldots \wedge \mathbf{not}(a@i) \\ 1 & \text{otherwise} \end{cases}$$

Thus,

- $(a-1)@0 = a@0 \oplus 1 = \mathbf{not}(a@0)$
- $$\begin{aligned} (a-1)@i = \quad & a@i \oplus 1 \oplus carry_{i-1} = \\ & \mathbf{not}(a@i) \oplus \mathbf{not}(\mathbf{not}(a@0) \wedge \mathbf{not}(a@1) \wedge \ldots \wedge \mathbf{not}(a@(i-1)) = \\ & \mathbf{not}(a@i) \oplus (a@0 \vee a@1 \vee \ldots a@(i-1)) \text{ for } 0 < i < n \end{aligned}$$

3.3.3 Kripke Structures

In the previous sections, we have shown how to represent the state space of an AUTOFOCUS system by boolean variables (the bits of the state variables). Besides, we have given a way to translate the terms $I(s)$ and $T(s, s')$ for the initial states and the transition relation to boolean formulas that are defined on the bits of the state variables (rather than their integer values) and in addition only contain the constants 0 and 1 and the operators $\mathbf{not}$, $\vee$, $\wedge$, $\Rightarrow$ and $\Leftrightarrow$.

Such a construction (a boolean state space together with I and T) is called a *Kripke structure*. A Kripke structure is often the (low-level) input for model checking algorithms and verification algorithms, as symbolic model checking using BDDs [McM93], and thus could also be used to directly integrate such model checking algorithms into the AUTOFOCUS/Quest tool (without the detour of using the input language of a certain tool like SMV).

3.4 Finding Solutions to Boolean Formulas

As pointed out in section 3.1.4, we can describe all possible execution sequences of a system by the term

$M = I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \ldots T(s_{k-1}, s_k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_i + 1)$,

that we derived from the state transition graph model of the system by unfolding the transition relation. If we eliminate user-defined data types and functions from I and T (see section 3.2) and represent the state space by boolean variables (see section 3.3), M will become a propositional formula.

To determine a particular execution sequence, we now just have to determine a solution for the propositional formula M. This can be done with algorithms and concepts belonging to the field of propositional logic which we describe in this section. The solution can then easily be translated back to an execution sequence — we simply have to combine the bits of the state variables again to determine their integer values, and then reverse the translation of AUTOFOCUS data types given in section 3.2. This is possible because each value of an AUTOFOCUS data type was mapped to a unique integer value. If there is no solution to M, then there is no execution sequence of the system of length $k+1$.

For the understanding of how to calculate solutions of propositional formulas, we assume basic knowledge of propositional logic. The reader not interested in this topic or not familiar with propositional logic can simply assume that it is possible to calculate a solution for a boolean formula and can safely skip this part. However, it is important to note that this is by far the most critical part concerning performance, it is the only task to be performed for specification based test sequence determination that has exponential complexity (see the comments about performance in section 3.4.4).

From this point on, we will abstract from the underlying AUTOFOCUS model and consider the general problem of solving propositional formulas. More precisely, our task is the following: we have a propositional formula f, which can be

- a boolean variable, originally standing for a bit of a state variable of our system in a certain execution step. We simply denote the variables by $A, B, \ldots$, as we do not need to consider the underlying AUTOFOCUS model anymore.
- the boolean constants 1 and 0 (resp. `True` and `False`).
- an expression of the form $\mathbf{not}(g), g \wedge h, g \vee h, g \Rightarrow h, g \Leftrightarrow h$, where g and h are boolean formulas.

Now, we are searching for an assignment to the variables $A, B, \ldots$, so that the formula evaluates to `True`. The algorithm should either give us such an assignment or tell us that there is none (i.e., the formula is "unsatisfiable").

This is a common problem belonging to the field of propositional logic. For the understanding of how it can be solved, we assume basic knowledge about this area. A good introduction to propositional logic can be found in [Fit90].

To find a solution for the boolean formula f, we first apply a series of conversions:

- simplify f
- convert f to negation normal form (NNF)
- convert f to conjunctive normal form (CNF)

All these conversions preserve the satisfiability of f: if there is a solution for the original formula, there is also a solution for the converted form.

Finding solutions for a formula in conjunctive normal form is called the "satisfiability problem" (SAT). There are various algorithms for SAT. We will explain the Davis-Putnam procedure that is implemented in the program SATO [Zha97], which we used in our tool to determine execution sequences.

3.4.1 Simplification

The formula is simplified by applying the following boolean algebra rules wherever possible (see [Bar91]):

- $a \wedge a \equiv a, a \wedge \mathbf{not}(a) \equiv \mathbf{not}(a) \wedge a \equiv 0, a \wedge 1 \equiv 1 \wedge a \equiv a, a \wedge 0 \equiv 0 \wedge a \equiv 0$
- $a \vee a \equiv a, a \vee \mathbf{not}(a) \equiv \mathbf{not}(a) \vee a \equiv 1, a \vee 1 \equiv 1 \vee a \equiv 1, a \vee 0 \equiv 0 \vee a \equiv a$
- $a \Rightarrow a \equiv 1, a \Rightarrow 0 \equiv \mathbf{not}(a), 1 \Rightarrow a \equiv a, a \Rightarrow 1 \equiv 1, 0 \Rightarrow a \equiv 1$
- $a \Leftrightarrow a \equiv 1, a \Leftrightarrow 1 \equiv 1 \Leftrightarrow a \equiv a, a \Leftrightarrow 0 \equiv 0 \Leftrightarrow a \equiv \mathbf{not}(a)$
- $\mathbf{not}(\mathbf{not}(a)) \equiv a, \mathbf{not}(0) \equiv 1, \mathbf{not}(1) \equiv 0$

I.e., if there is a subterm of the form $a \wedge a$ in the formula, we replace it by a, etc. The simplified formula is logically equivalent to the original formula, meaning that for all possible valuations of the variables, the result is the same.

Note that the simplified formula does not contain constants (0 or 1) anymore. The only exception to that is if the simplified formula is "0" or "1" itself — but in this case we are finished: if it is "0" (`False`), the formula is unsatisfiable, and if it is "1" (`True`), we can just choose any valuation, as the formula always evaluates to `True`.

The simplification of boolean formulas has also a very important effect regarding our translation of AUTOFOCUS system models: it saves us in most cases from complicated optimisations at earlier stages of the translation. Consider the constructor term `Valid(CI(1, 0),1)` of type `Card` (our example in section 3.2). Using the bitfield encoding (see section 3.2.4), this term is translated to

$0 + (0 + 1 \cdot 1 + 0 \cdot 2) \cdot 2 + 1 \cdot 8$

Obviously, this expression can be simplified (by replacing it with "10"). However, we do not need to carry out the simplification at this stage. When the expression is later separated into its bits (as shown in section 3.3.1), the terms for the bits only consist of constants 1 or 0. Now, when we apply the above simplifications of boolean formulas, the values of the bits are actually "calculated", and the result is the same as if we had replaced the expression with "10" before.

3.4.2 Conversion to NNF

In the next step, we have to convert the formula to negation normal form (NNF). A formula is in NNF if the negations $\mathbf{not}(\ldots)$ only occur in front of the variables. This is easy, we just have to apply some boolean rules again (see [Fit90]):

- $\mathbf{not}(a \wedge b) \equiv \mathbf{not}(a) \vee \mathbf{not}(b)$
- $\mathbf{not}(a \vee b) \equiv \mathbf{not}(a) \wedge \mathbf{not}(b)$
- $(a \Rightarrow b) \equiv \mathbf{not}(a) \vee b$
- $\mathbf{not}(a \Rightarrow b) \equiv \mathbf{not}(\mathbf{not}(a) \vee b) \equiv a \wedge \mathbf{not}(b)$
- $\mathbf{not}(a \Leftrightarrow b) \equiv a \Leftrightarrow \mathbf{not}(b)$
- $\mathbf{not}(\mathbf{not}(a)) \equiv a$

Note that after application of these rules, the operator $\Rightarrow$ does not occur in the formula anymore. We could also have eliminated the operator $\Leftrightarrow$, but we will do that in the next step, together with the conversion to CNF.

3.4.3 Conversion to CNF

Finally, we convert the formula to conjunctive normal form (CNF). A formula is in CNF if it is a conjunction of clauses, where a clause is a disjunction of literals. A literal is either a variable or its negation. For example,

$(A \vee \mathbf{not}(B) \vee C) \wedge (B \vee D) \wedge (\mathbf{not}(A) \vee D \vee E)$

is in CNF.

The standard algorithm (see for example [Fit90]) to convert a formula in NNF to CNF is repeatedly applying the distributive law

$f \vee (g \wedge h) \equiv (f \vee g) \wedge (f \vee h)$.

However, the size of the result can be exponential with respect to the size of the original formula, which makes it impossible to find solutions even for medium sized formulas. To avoid this problem, we use another approach outlined in [PG86].

The idea of this approach is to replace subexpressions by newly generated variables and to add conditions to the CNF stating that the new variables and the corresponding subexpressions must always have the same value (i.e., they must be equivalent).

This way, we convert a formula f into its conjunctive normal form $\mathsf{CNF}(f)$ as follows:

- If f is a literal, then $\mathsf{CNF}(f) = f$.
- If f is of the form $g \wedge h$, then $\mathsf{CNF}(f) = \mathsf{CNF}(g) \wedge \mathsf{CNF}(h)$
- If f is of the form $g \vee h$, then
  - Check if f is already in CNF, in this case $\mathsf{CNF}(f) = f$.
  - Otherwise, generate new variables v_g and v_h for g and h. The result is $\mathsf{CNF}(f) = v_g \vee v_h$.
    As the result now contains new variables, at the end of the translation we also have to add the corresponding conditions to the CNF that make sure that v_g and v_h are equivalent to g and h — these conditions are calculated when generating the new variables (see below).
- If f is of the form $g \Leftrightarrow h$, generate new variables v_g and v_h for g and h. The result is
  $\mathsf{CNF}(f) = (\mathbf{not}(v_g) \vee v_h) \wedge (v_g \vee \mathbf{not}(v_h))$ (which is equivalent to $v_g \Leftrightarrow v_h$).

Now we show what has to be done to generate a new variable v_g for the subexpression g:

- If g is a literal, there is no need to do anything, just let $v_g = g$.
- If g is of the form $a \circ b$ ($\circ$ being one of the operators $\wedge, \vee, \Leftrightarrow$), then:
  - generate a new variable v_g for g
  - generate new variables v_a and v_b for a and b (by recursively applying the algorithm for generating new variables)
  - If g is of the form $a \wedge b$ add the condition
    $(\mathbf{not}(v_g) \vee v_a) \wedge (\mathbf{not}(v_g) \vee v_b) \wedge (v_g \vee \mathbf{not}(v_a) \vee \mathbf{not}(v_b)$
    to the CNF. This is equivalent to $v_g \Leftrightarrow (v_a \wedge v_b)$.
  - If g is of the form $a \vee b$, then add
    $(v_g \vee \mathbf{not}(v_a) \wedge (v_g \vee \mathbf{not}(v_b)) \wedge (\mathbf{not}(v_g) \vee v_a \vee v_b)$,
    which is equivalent to $v_g \Leftrightarrow (v_a \vee v_b)$.
  - If v_g is of the form $v_a \Leftrightarrow v_b$ then add
    $(v_g \vee v_a \vee v_b) \wedge (\mathbf{not}(v_g) \vee \mathbf{not}(v_a) \vee v_b) \wedge (\mathbf{not}(v_g) \vee v_a \vee \mathbf{not}(v_b)) \wedge (v_g \vee \mathbf{not}(v_a) \vee \mathbf{not}(v_b))$,
    equivalent to $v_g \Leftrightarrow (v_a \Leftrightarrow v_b)$.

This algorithm can be optimized further (e.g. by treating expressions with multiple operands, like $a \wedge b \wedge c \wedge d$ in a special way), but for reasons of brevity we will not go into details here. See [PG86] for more explanation.

The result is a boolean formula in CNF with a size linear with respect to the size of the original formula. The formula in CNF is not logically equivalent to the original formula (as it contains new variables), but if the original formula is satisfiable, so is the formula in CNF. The only difference is that in the solution to the formula in CNF we additionally have the values for the new variables, which we can simply ignore.

3.4.4 The Davis-Putnam Procedure

Now that we have a formula in CNF, we can try to find a solution (i.e., a valuation of the variables that satisfies the formula). This is called the "satisfiability problem" (SAT). Various algorithms exist to solve the satisfiability problem — we explain the Davis-Putnam procedure implemented in the tool SATO [Zha97], which we used as an external program in our execution sequence determination. Most tools for the SAT problem take a standardized form ("DIMACS") of a formula in CNF as an input, so it would be very easy to switch to other programs (like PROVER [Bor97], an implementation of the Stålmarks method [Stå89]). The main advantage of SATO is that it is freely available, and its performance is sufficient to serve our purposes.

As the aim of this thesis is to describe ways to determine test data and not to go into details of proof procedures in propositional logic, we give an more intuitive description of the Davis-Putnam procedure than is usually found in the literature. For a formal explanation and a proof of its correctness, see [Fit90].

The Davis-Putnam procedure [DP60] searches for a solution to a formula in CNF by choosing variables and fixing their values to 0 or 1. This way, it creates a decision tree (see the example in Fig. 3.17). In the worst case, both values have to be tried at the same time, so we have a branch in the tree (see the first step in the example). On the other hand, for reasons explained later, we can often restrict our search to only one of the values (see the following steps).

After we have fixed a variable to a value, we simplify the resulting formula using boolean rules (see 3.4.1). Then we again take a leaf in the tree (where some variables have already been fixed to particular values) and try values for one of the remaining variables etc.

At some point, there could be a leaf node in the tree where we already fixed some variables, that is simplified to 1 (`True`). Then we have found a valuation for the variables that satisfies the formula (if there are any variables the value of which we did not fix, they can have an arbitrary value). On the other hand, if all leaf nodes of the tree are 0 (`False`), it is not possible to find a solution, so the formula is unsatisfiable.

After this informal introduction, we can now give the Davis-Putnam procedure to find a solution for the boolean formula f (which must be in CNF). First, we have to explain some notations:

- Let L be a literal (i.e., a variable or its negation). We denote with $\overline{L}$ the *complement* of L, so that for a variable V, $\overline{V} := \mathbf{not}(V)$, and $\overline{\mathbf{not}(V)} := V$.

- For a formula g in CNF and a literal L contained in g, we denote with $g[L \leftarrow 0]$ resp. $g[L \leftarrow 1]$ the result of fixing the value of L to 0 resp. 1 in the formula g and simplifying it. Note that in $g[L \leftarrow 1]$, all clauses of g that contain L have been removed, as well as all occurrences of $\overline{L}$ in the clauses. This is because of the boolean rules $1 \vee a \equiv 1$, $0 \vee a \equiv a$ and $1 \wedge a \equiv a$.

The Davis-Putnam procedure works as follows:

- Start with a decision tree with f as the root node.

- Apply one of the following rules, until there is either a leaf node marked with "1" (then we have found a solution), or all leaf nodes are marked with "0" (then the formula is unsatisfiable):

  - **Splitting Rule** Choose a leaf node in the tree with a formula g containing the literal L. Create two successor nodes of this node containing $g[L \leftarrow 0]$ and $g[L \leftarrow 1]$.
    The application of this rule corresponds to trying both values for a literal, i.e. there is a solution for g if there is a solution for $g[L \leftarrow 0]$ or a solution for

Step 0

$$(\mathbf{not}(B) \vee C \vee \mathbf{not}(D)) \wedge$$
$$(A \vee \mathbf{not}(C)) \wedge (A \vee C \vee D) \wedge$$
$$(A \vee B) \wedge (\mathbf{not}(A) \vee \mathbf{not}(B) \vee \mathbf{not}(C))$$

Step 1
(splitting rule)

$B \leftarrow 0$

$$(\mathbf{not}(0) \vee C \vee \mathbf{not}(D)) \wedge$$
$$(A \vee \mathbf{not}(C)) \wedge (A \vee C \vee D) \wedge$$
$$(A \vee 0) \wedge (\mathbf{not}(A) \vee \mathbf{not}(0) \vee \mathbf{not}(C))$$
$$=$$
$$(A \vee \mathbf{not}(C)) \wedge (A \vee C \vee D) \wedge (A)$$

$B \leftarrow 1$

$$(\mathbf{not}(1) \vee C \vee \mathbf{not}(D)) \wedge$$
$$(A \vee \mathbf{not}(C)) \wedge (A \vee C \vee D) \wedge$$
$$(A \vee 1) \wedge (\mathbf{not}(A) \vee \mathbf{not}(1) \vee \mathbf{not}(C))$$
$$=$$
$$(C \vee \mathbf{not}(D)) \wedge (A \vee \mathbf{not}(C)) \wedge$$
$$(A \vee C \vee D) \wedge (\mathbf{not}(A) \vee \mathbf{not}(C))$$

Step 2
(one-literal rule)

$C \leftarrow 1$

$$(1 \vee \mathbf{not}(D)) \wedge (A \vee \mathbf{not}(1)) \wedge$$
$$(A \vee 1 \vee D) \wedge (\mathbf{not}(A) \vee \mathbf{not}(1))$$
$$=$$
$$(A) \wedge (\mathbf{not}(A))$$

Step 3
(one-literal rule)

$A \leftarrow 0$

$$(0) \wedge (\mathbf{not}(0))$$
$$=$$
$$0$$

Step 4
(one-literal rule)

$A \leftarrow 1$

$$(1 \vee \mathbf{not}(C)) \wedge (1 \vee C \vee D) \wedge (1)$$
$$=$$
$$(\mathbf{not}(C)) \wedge (C \vee D)$$

Step 5
(one-literal rule)

$C \leftarrow 0$

$$(\mathbf{not}(0)) \wedge (0 \vee D)$$
$$=$$
$$(D)$$

Step 6
(one-literal rule)

$D \leftarrow 1$

$$1$$

$\Rightarrow$ **Solution found — valuation is**
$[A \leftarrow 1, B \leftarrow 0, C \leftarrow 0, D \leftarrow 1]$

Figure 3.17: Davis-Putnam Procedure Example

$g[L \leftarrow 1]$. As this increases the number of cases we have to consider, the splitting rule must only be applied if we cannot apply any other rule.

– **One-Literal Rule** If we have a leaf node in the tree with a formula g that

contains a clause consisting of only one literal L, create a successor node of this node containing $g[L \leftarrow 1]$.
Note that in this case, we do not have to consider $g[L \leftarrow 0]$: g is of the form $L \wedge C_1 \wedge C_2 \wedge \ldots$, where $C_1, C_2, \ldots$ are the other clauses of g. Because of $0 \wedge \ldots \equiv 0$, there cannot be a solution with $L = 0$.

– **Pure-Literal Rule** If we have a leaf node in the tree with a formula g containing a literal L but not its complement $\overline{L}$, create a successor node of this node containing $g[L \leftarrow 1]$.
This is possible because the clauses in $g[L \leftarrow 1]$ are a subset of the clauses in $g[L \leftarrow 0]$ (in $g[L \leftarrow 1]$ we have deleted all clauses containing L, and in $g[L \leftarrow 0]$, we have deleted all occurrences of L in the clauses). Thus, if we do not find a solution for $g[L \leftarrow 1]$, $g[L \leftarrow 0]$ will also be unsatisfiable, because the clauses are connected with $\wedge$, and $g[L \leftarrow 0]$ only contains more clauses. Therefore we can restrict our search to $g[L \leftarrow 1]$.

– **Subsumption Rule** If we have a leaf node in the tree with a formula g containing two clauses C_1 and C_2, and the literals in C_1 are a subset of the literals in C_2, we can remove C_2 from the formula.
This is because if there is a solution for g, C_1 must evaluate to 1. As C_2 contains the same literals as C_1 plus others connected with $\vee$, C_2 will automatically evaluate to 1 (because of $1 \vee a \equiv 1$).

Note that the Davis-Putnam procedure only has to consider the leaf nodes of the tree. In the more formal literature, the set of leaf nodes is referred to as a “block”, and the rules transform a block by changing a formula in the block or splitting it into two formulas.

Complexity

The Davis-Putnam procedure is by far the most performance critical part of our approach to determine test data for systems specified with AUTOFOCUS. All transformations described so far (i.e., creating terms for initial states and transition relation, eliminating hierarchical data types, separating the state variables into their bits etc.), can be carried out in polynomial time. However, finding a solution for a formula in CNF (the “satisfiability problem”) is NP-complete.

This means that so far there is no algorithm with a worst case execution time better than $O(2^n)$ in the number of variables — which essentially corresponds to trying all possible valuations.

That is also true for the Davis-Putnam procedure, but it was designed such that in most cases a solution is found very fast. It is important to mention that the efficiency of the Davis-Putnam procedure depends very much on how the rules are applied (as there is often a choice which rule to apply, which literal to use for splitting, etc.). Thus, we need good heuristics.

The tool SATO is a very efficient implementation of the Davis-Putnam procedure using heuristics based on extensive expert knowledge. Therefore (and because SATO is implemented in C rather than Java), we use SATO as an external tool in our program to determine execution sequences instead of implementing the Davis-Putnam procedure ourselves.

3.5 Conclusion

In this chapter, we have described how to model the behaviour of systems specified with AUTOFOCUS using logical formulas. Based on the concepts and algorithms we presented to calculate execution sequences of a certain length, we are now prepared to determine test data for system testing (by adding requirements to the formulas, e.g. specifying that a particular transition should be executed or an automaton of the system should be put into a certain control state).

To determine an execution sequence, we had to perform the following steps:

- transform the AUTOFOCUS system description into a state transition graph by determining its state space and formulas for the initial states and the transition relation (see section 3.1),
- to specify an execution sequence, create copies of the state variables for each step and unfold the transition relation, which gives the formula $M = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$,
- eliminate user-defined data types and functions from the formulas (see section 3.2),
- separate the state variables into bits and translate the formulas accordingly (see section 3.3),
- convert M to CNF and apply SATO to find a solution (see section 3.4)
- if there is a solution, translate the solution back to an execution sequence by recombining the bits and reversing the translation of the data types — if there is no solution, there is no execution sequence of length $k + 1$.

Moreover, we pointed out how to use these concepts to translate an AUTOFOCUS system description to the SMV input language or to a Kripke structure which can be used to interface to low-level model-checking algorithms.

This concludes our description of the algorithms we need for our procedure to determine execution sequences. The rest of this thesis is concerned with applications of this procedure for system testing and its evaluation. In particular, we explain different kinds of test cases and how they can support system testing.

Chapter 4

Specification-based Test Sequence Determination

The aim of the AutoFocus/Quest tools is to facilitate the design of safety-critical embedded systems by providing user-friendly graphical specification methods in connection with supporting formal verification of correctness criteria, testing, simulation and code generation. For this purpose, interfaces between AutoFocus/QUEST and external tools like the model checker SMV [McM92] or the Verification Support Environment VSE [RSW97] have been developed.

Our work provides a connection from AutoFocus/QUEST to satisfiability solvers like SATO, making it possible to determine execution sequences from a specification by solving a formula in propositional logic. In this chapter, we show how to apply this method for the purpose of system testing and verification by using these execution sequences to obtain input and output data for system tests. This is done by specifying *test cases*, which describe particular desired behaviours of a system. For example, a test case could be that a certain sequence of transitions should be executed or a particular control state in one of the automata should be reached. We call the various ways to specify test cases (e.g. by giving transition sequences or control states to be reached) *test scenarios*. With the help of the logical model of the specification, we can now calculate input and output data of a particular execution that fulfills the requirements given by the test case. The input and expected output data is then used to test the behaviour of the implementation for the corresponding test case. Thus we also call the calculated input and output data a *test sequence* and our test strategy "*specification based test sequence determination*".

An overview of testing and verification methods for systems specified with AutoFocus/Quest is given in [SK98] (in German) and [KMSS98]. As we will see, specification-based test sequence determination can in many cases support or replace the test methods described in these papers. Based on the concepts explained in these papers and our own ideas, we present selected test scenarios where specification-based test sequence determination can be employed successfully.

Bounded Model Checking (BMC) is a generalization of specification based test sequence determination that can be used as an efficient alternative to conventional model checking to verify properties of a specification expressed in temporal logic, if we are only interested in counterexamples of a limited length. In addition to the algorithms needed for test sequence determination, we also implemented a bounded model checker for AutoFocus in our work and will present its general concepts in this chapter.

The chapter concludes with thoughts about non-deterministic test sequences, which are an important problem to consider when testing systems specified using AutoFocus.

4.1 System Testing Using Test Sequences

In this section, we show how systems can be tested by determining test data (input and output values) from the specification, using our logical model. In the following sections, we will then describe six specific test scenarios where this method can be applied successfully and show in which way it can be used to support system testing.

4.1.1 Test Sequences

In our model, the implementation of a system interfaces to the environment via a set of input and output ports (see Fig. 4.1). Its execution is controlled by a global clock, and at each clock tick, certain data values are presented on the output ports as a response to the input data we gave to the implementation. We usually have no knowledge about the internal structure of the system we want to test, in particular there does not need to be a relation to the internal structure of an AUTOFOCUS specification. The implementation could for example be a hardware circuit in an embedded system, the main application area of AUTOFOCUS.

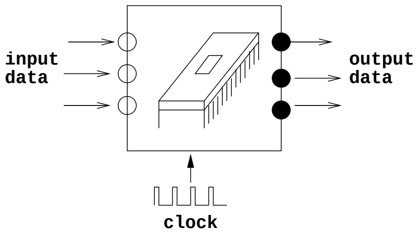


Figure 4.1: Implementation of a System

Such an implementation can be tested using **test sequences** (see Fig. 4.2). Test sequences describe the communication behaviour of the implementation for a certain test case. Thus, a test sequence consists of a number of steps, corresponding to the clock ticks. For each step, the test sequence contains

- the **input data** we have to present to the input ports of the implementation, and
- the expected **output data** at its output ports.

Test sequences can be graphically depicted by EETs (see section 2.6). To test the implementation, we pass to its input ports at each step the input data specified in the test sequence and check if the data values at the output ports are as expected.

Test sequences are used to verify if the system is correct with respect to a particular behaviour. As an example, consider the simple system `SimpleSys` again (see Fig. 4.3). This system should always respond with `Msg(Present)` at the output port `o` when it has read `Msg(Present)` from the input port `i` in the step before (corresponding to the execution of the transition in the specification). To test this behaviour, we can now pass the value `Msg(Present)` to the system for a number of execution steps and check if the result at the port `o` is correct.

Fig. 4.4 shows a 4-step test sequence for `SimpleSys` that can be used to verify the behaviour described above (i.e., `Msg(Present)` is passed to the implementation for the first three steps and we expect `Msg(Present)` to be output at port `o` at the next step).

4.1.2 Determining Test Sequences from a Specification

Though it was straightforward to determine a test sequence for the simple system showed above, finding input data and the expected output data for system tests usually a difficult

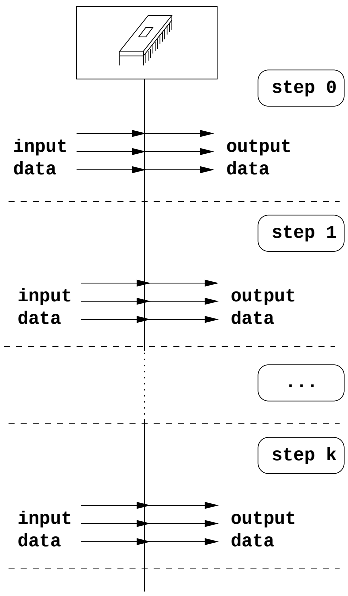


Figure 4.2: Test Sequence

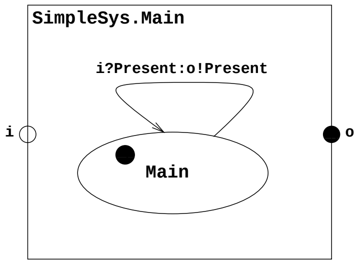


Figure 4.3: Simple System

task. Consider for example the `Till` component of the cash point system (see Fig. 2.1), which has 5 input and 4 output ports. How could we find input data to test the behaviour of the system for a certain test case, for instance to test if the behaviour is correct when a timeout happens on entering a PIN ?

This problem can be solved by **specification-based test sequence determination**. In addition to the implementation, we now have a specification of the system to be tested, as an AUTOFOCUS system model. The internal structure of the implementation does not necessarily have to correspond to that of the specification, and we usually cannot make any statements about its internal state — this is because firstly we probably do not have access to this data (unless we have implemented some kind of trace mechanism), and secondly, because the system might have been implemented in a completely different way than described in the specification. We only require that the interface (i.e., the input and output ports of the system from/to the environment) has to be the same for the specification and the implementation.

The specification is by definition assumed to be correct and is used to describe all possible behaviours of the implementation. We can now make use of the methods described in the last chapter to determine execution sequences from an AUTOFOCUS system model to calculate test sequences for particular test cases. This is done by specifying a test case (i.e. the case that a timeout occurs in the component `Till`) and then determining an execution

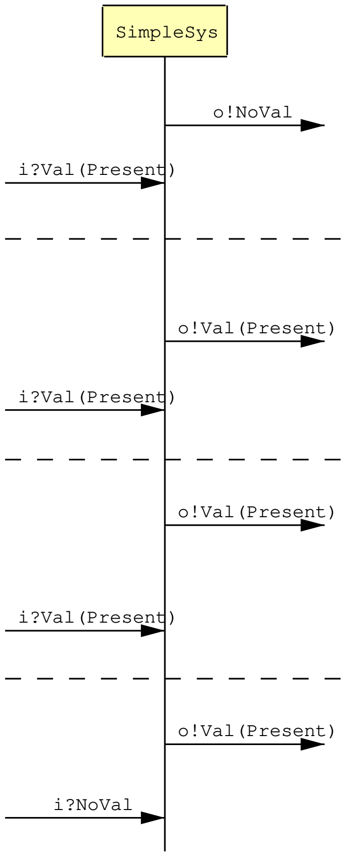


Figure 4.4: Test Sequence for Simple System

sequence from the specification that corresponds to one possible behaviour fulfilling the requirements of the test case (i.e. one possible behaviour of the specification where a timeout occurs in the component `Till`). Such an execution sequence is calculated by solving a propositional formula using SATO, as described in the previous chapter. Finally, we take the input and output data from the execution sequence and use it to test the system. In detail, the following steps have to be performed (see Fig. 4.5):

1. Translate the desired test case to a predicate P on the state variables (ports, control states and local variables) at the different execution steps from step 0 to a maximum step k. For example, to test the behaviour of the `Till` component in case of a timeout, we have to specify that at one of the steps, the value `ErrorTimeout` has to be output at the port `Display`. Thus, the predicate would be:
$P = (\mathbf{Display}.0 == \mathbf{ErrorTimeout}) \vee (\mathbf{Display}.1 == \mathbf{ErrorTimeout}) \vee \ldots \vee$
$(\mathbf{Display}.k == \mathbf{ErrorTimeout})$.
The corresponding test sequence has the length $k + 1$.

2. Generate the logical formula M, describing all possible executions of length $k + 1$ of the system as a predicate on the state variables in the steps (where I and T are the semantics of the model, see section 3.1.4):
$M = I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \ldots T(s_{k-1}, s_k)$.

3. The execution sequences corresponding to the desired test case are now specified by

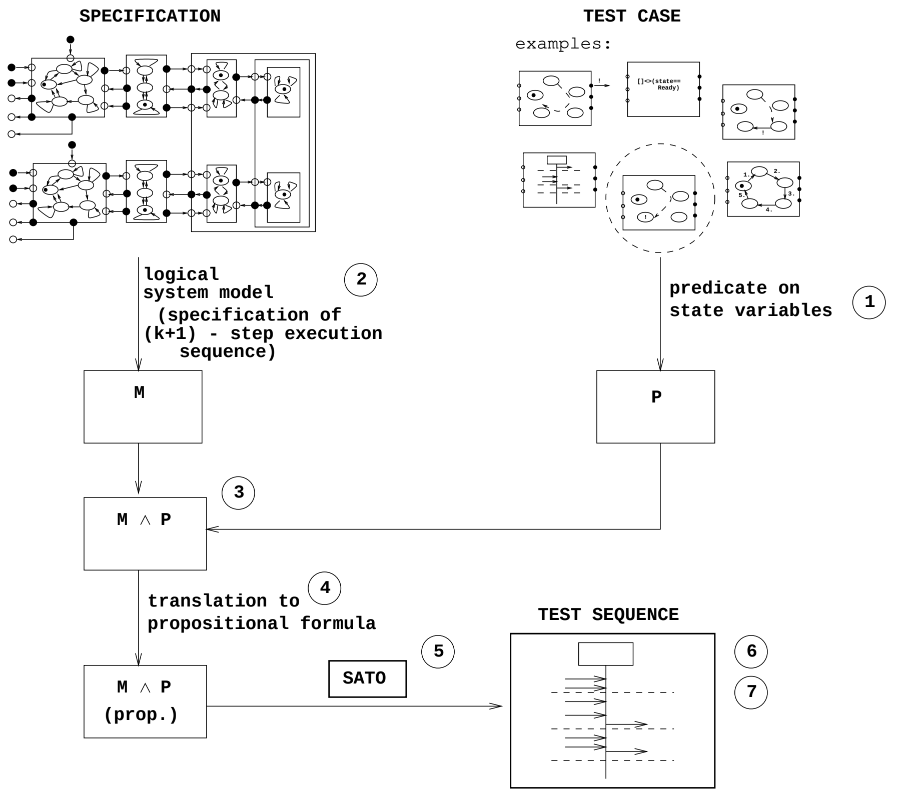


Figure 4.5: Specification Based Test Sequence Determination

the formula
$M \wedge P$.

4. Translate $M \wedge P$ to a propositional formula by eliminating user-defined data types and functions and by encoding the state space using boolean variables (see sections 3.2 and 3.3).

5. Use SATO to calculate an execution sequence, as a solution to $M \wedge P$ (see section 3.4). The solution is an example of a behaviour of the system corresponding to the test case (i.e. in the example, an execution sequence where `ErrorTimeout` is output at some step at the port `Display`).

6. If SATO did not find a solution, the system behaviour we described is impossible. Otherwise, we take the input and output data from the calculated execution sequence and generate an EET that describes the desired test sequence.

7. Test the implementation by passing to it the input data we calculated at each step and checking if the output data is as expected.

In the following, we describe interesting test scenarios (i.e., ways to specify test cases). The corresponding test sequences are always derived from the specification and our target is to show if the implementation behaves as we specified by passing the calculated input data to it. The scenarios we explain are:

- execution of a particular **transition sequence** in one of the automata of the specification,

- execution of a **single transition** in one of the automata,
- putting an automaton into a particular **control state**
- finding a test sequence so that specified **output data** appears at one of the output ports (e.g. an error message)
- testing a behaviour corresponding to a given **EET**
- testing a behaviour specified by a formula given in temporal logic (**bounded model checking**)

4.1.3 Non-Deterministic Test Sequences

Before describing the test scenarios, we have to mention an important problem we have to face when testing systems with the help of test sequences: non-deterministic systems.

Consider the system specification in Fig. 4.6. The component `NondetSys.Main` only has one output port `o` and a STD assigned to it with two transitions. Suppose we want to find a test sequence so that "1" is output at the port `o` at a certain step. SATO will easily calculate such a test sequence for us — there is no input data, and the sequence of the values output at port `o` can be any combination of "0" or "1".

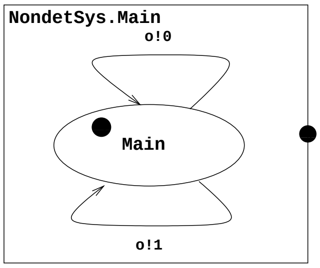


Figure 4.6: A Non-Deterministic System

However, the implementation is free to decide which transition to execute, as always both transitions can be executed. Thus, the behaviour of the implementation will almost certainly differ from the behaviour expected when regarding the calculated test sequence. The implementation could even take the transition "`o!0`" at all times, so the value "1" would never be output.

This unfavourable effect of non-determinism can be even more subtle. For example, a non-deterministic choice could result in different possible values for local variables, which might only influence the result in much later steps.

We can only detect that the behaviour of a system deviates from our calculated test sequence if we have access to all internal channels, local variables and control states of the automata in the implementation (which thus must have the same internal structure as the specification). As explained at the beginning of this chapter, we can usually not assume that this is possible.

Deterministic System Specifications

To avoid these problems, we assume that the specification we determine test sequences from is deterministic.

An AUTOFOCUS specification is deterministic, if at no time during the execution of the system two transitions in an automaton are enabled at the same time.

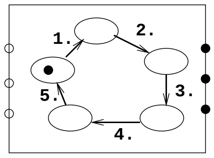
This definition is sufficient because the transitions themselves do not contain non-determinism (they write fixed values to the output ports and local variables).

Specifically, a system is deterministic if the preconditions for all transitions leading out of one particular control state in an automaton contradict each other. For example, the `Till` component (see Fig. 2.3) is deterministic: from state `Ready`, there are two transitions, one can only be executed when there is a data value at port `Slot` (transition `(0) Slot?card:...`) and the other only when there is no data value (transition `(1) Slot?:...`). Similar arguments apply to the other control states of the `Till` component.

If we have a deterministic specification, the behaviour of the system is fixed. For a particular sequence of input values for the ports, there is only one possible execution of the system (given by the execution of the transitions in the automata), and one possible sequence of output values. Thus, we can test the implementation by passing the input data to it and comparing the output data with the output data in the test sequence we derived from the specification.

In the following, we will concentrate on test strategies for deterministic specifications. At the end of this chapter, we will present some thoughts about how to handle non-deterministic specifications (see 4.8).

4.2 Determining Input Data for Transition Sequences



One scenario our approach is particularly suitable for is determining input data for transition sequences. Suppose for an automaton in our specification we have a sequence of transitions and we want to simulate the execution of these transitions in the implementation and check if it responds correctly.

With the algorithms for test sequence determination we developed, we can check if a particular transition sequence is executable at all, and if so, what input data we have to give to the implementation to enforce its execution and what data is expected as output.

We can calculate such a test sequence as follows: In section 3.1.2, we showed how to translate a transition $trans_i$ into a term $T_{trans_i}(s, s')$ that specifies that this transition is executed (i.e., it evaluates to `True`, if the transition $trans_i$ is executable in state s and s' is the corresponding successor state). So, if we have the transition sequence $trans_1$, ..., $trans_k$ for an automaton in the specification, then the predicate P specifying the execution of that transition sequence is

$P = T_{trans_1}(s_0, s_1) \wedge T_{trans_2}(s_1, s_2) \wedge \ldots \wedge T_{trans_k}(s_{k-1}, s_k)$
(see Fig. 4.7 for an illustration).

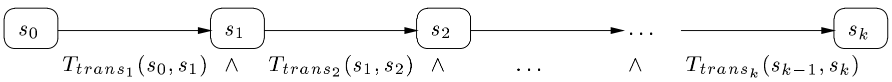


Figure 4.7: Specifying a Transition Sequence

To calculate the test sequence, we now determine the formula M, describing all possible executions of the system, convert $M \wedge P$ into a propositional formula and then calculate a solution, as explained above. However, we have to take care that if the system consists of more than one component (i.e., the automaton we specified the transition sequence for is only a part of the system), we substitute the port names in the $trans_i$ terms as shown in section 3.1.3 so they correspond to the port names that appear in the formula M.

Optimisation

If we look at $M \wedge P$, we can see that an optimisation is possible. Remember that

$$M = I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \ldots T(s_{k-1}, s_k),$$

where $I(s)$ stands for the term for the initial states and the $T(s_i, s_{i+1})$ stand for the term for the transition relation (see section 3.1). Note that for each automaton, the $T(s_i, s_{i+1})$ contain a disjunction of all the T_{trans_j} for the transitions of the automata and the idle transition, meaning that either one of these transitions or the idle transition must be executed. As we specified with the predicate P a particular transition $trans_i$ that must be executed, we can omit the T_{trans_j} for the other transitions and the idle transition in the corresponding automaton. If we denote this modification of $T(s_i, s_{i+1})$ as $(T(s_i, s_{i+1}))_{trans_j}$, we can also write $M \wedge P$ as

$$M \wedge P = I(s_0) \wedge (T(s_0, s_1))_{trans_1} \wedge (T(s_1, s_2))_{trans_2} \wedge \ldots \wedge (T(s_{k-1}, s_k))_{trans_k}$$

If the system consists only of one automaton (which is a very common case for determining input data for transition sequences), we can in fact omit all other transitions, and $M \wedge P$ becomes

$$M \wedge P = I(s_0) \wedge T_{trans_1}(s_0, s_1) \wedge T_{trans_2}(s_1, s_2) \wedge \ldots \wedge T_{trans_k}(s_{k-1}, s_k)$$

As the formula $M \wedge P$ now only consists of the terms for the transitions contained in the transition sequence, its size is fairly small, and thus with the given method, it can be solved quite efficiently (see chapter 6 for an evaluation).

4.2.1 Testing Arbitrary Transition Sequences

One application for test sequence determination for transition sequences is to determine input data for arbitrary transition sequences specified manually by the tester. Suppose the tester has a particular behaviour of the system in mind, corresponding to the execution of a transition sequence in one of the components in the specification. For example, for the `Till` component (see Fig. 2.3), we could specify that first a card is entered into the system, then a valid pin is typed in at the keypad, and then the card is rejected with the error "`DayLimitExceeded`". This corresponds to the execution of the three transitions (1)–(5)–(9), from state `Ready` to state `CardEntered`, then to state `PINEntered` and finally back to state `Ready` (see Fig. 4.8). A test sequence for this behaviour is shown in Fig. 4.9.

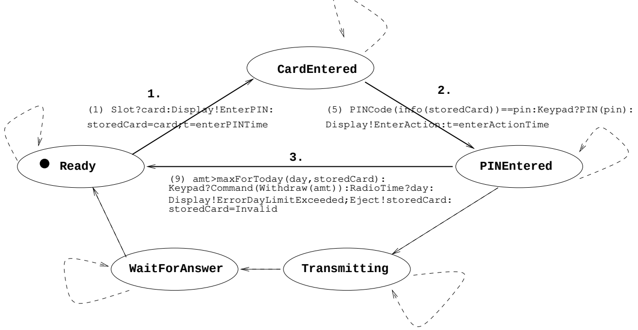


Figure 4.8: Transition Sequence for `Till`

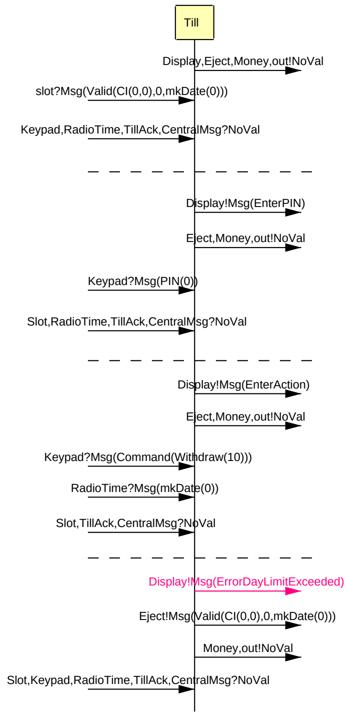


Figure 4.9: Test Data for Transition Sequence in Fig. 4.8

Our test sequence determination algorithm can help with the following problems:

- Check if such a transition sequence is possible at all in the specification – otherwise the test case is invalid.
- If the transition sequence is not executable, calculate an executable prefix of the transition sequence (i.e., show up to which point this transition sequence would be executable). This can be done by repeatedly removing the last transition of the sequence, or more efficiently by a bisection-like approach.
- Determine input data to pass to the implementation to force the behaviour corresponding to the execution of that particular transition sequence.
- Determine the expected output data, so the results output by the implementation can be verified.
- If we calculate all solutions to M instead of just one (this can be done by slightly modifying the Davis-Putnam procedure shown in section 3.4.4), we can determine *all* possible test sequences (differing in input data and output data) for this transition sequence. In case of a deterministic system, we can then feed the input data to the implementation and make sure the implementation behaves correctly *in all cases* when the particular transition sequence is executed.

Note that test sequence determination is also possible if the component we have a transition sequence for is part of a larger system. Thus, we can also use our method to find out how the surrounding components must behave in case a transition sequence is executed, and what input data we have to feed to these components to force the execution of a transition sequence in an inner component we might not have direct access to.

4.2.2 Determining Input Data for Transition Tours

A more systematic approach to use transition sequences for system testing is given in [SK98]. In this paper, the concept of a so-called "*Combined Function and Trace Test*" (CFTT) is presented. With the CFTT method, the correctness of an implementation is verified by selecting test cases for the transitions of the system. A test case tc_{ij} for a transition $trans_i$ is a predicate on the state space (for example, two test cases for the transition (1) from state `Ready` to state `CardEntered` in the `Till` component would be that the inserted card is either valid or invalid). It is assumed that the transitions behave either always correctly or incorrectly for a particular test case.

Now, a new automaton is determined from the original one that contains all the test cases for the transitions instead of the transitions themselves. The behaviour for the test case tc_{ij} for transition $trans_i$ is given by $T_{trans_i} \wedge tc_{ij}$. The states of the new automaton are derived by partitioning the states of the original automaton (for example, for `CardEntered` we could have two new states `ValidCardEntered` and `InvalidCardEntered`, corresponding to the effects of the two test cases for the transitions). For each test case of $trans_i$ it is calculated which of the new states can be the source and destination states for that test case (e.g., the test case where the inserted card is valid can only lead from `Ready` to the new state `ValidCardEntered` and not to `InvalidCardEntered`) [1].

Finally, a transition sequence for the new automaton is calculated that starts at the initial state, covers all test cases for the transitions, and ends at the initial state. This is called a "*transition tour*" and can be used to verify the behaviour of the system with respect to the test cases for the transitions.

What remains is to determine input data to pass to the implementation so that the transition tour can be executed. That can now be done with the help of SATO, as shown above.

For a more detailed explanation of the CFTT method, we refer the reader to [SK98].

4.2.3 Determining a Transition Tour without Partitioning

In the CFTT method, the partitioning of the states guarantees that a calculated transition tour is always executable. However, the partitioning is a difficult task that can in general not be performed automatically.

Unfortunately, when the states have not been partitioned, arbitrary transition sequences covering all transitions are in general not executable. This is because it depends on the execution of other transitions whether a certain transition is executable or not. As an example, consider the `Till` component again (Fig. 2.3). Transition (3), corresponding to a timeout occurring when the user was prompted for a PIN, is only executable after transition (2) has been executed often enough so that the value of the timer variable `t` became 0.

However, of course for most systems there are executable transition sequences covering all transitions. Such transition tours can be selected manually by the tester, who usually has sufficient knowledge about the system. Our method to determine input data for transition sequences can aid the tester in this task, by showing him if a certain transition sequence is executable, and if this is not the case, which is the first transition of the sequence that cannot be executed.

For the `Till` component, it is fairly easy to determine a transition tour manually: the system always reaches the state `Ready` again after execution of a number of transitions corresponding to a certain behaviour of the system (e.g. an invalid card was inserted, a timeout occurred, money was withdrawn successfully etc.). Thus, there is a transition tour consisting of loops from state `Ready` back to this state. We manually found such a transition tour (of length 34) for the `Till` component and verified that it is indeed executable.

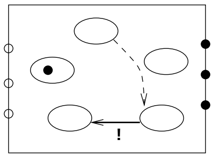
[1] In fact, it is even possible to use SATO and the approaches described in this thesis for this purpose: suppose $trans_i$ is a transition from state Q to state S. The elements (e.g. q_k and s_l) of the partitions of Q and S are characterized by predicates on the state space (e.g. `is_Valid(card)`). Thus, if we want to know if the test case tc_{ij} can lead from state q_k and s_l, we can check with SATO if there is a solution for
$q_k(s_0) \wedge T_{trans_i}(s_0, s_1) \wedge tc_{ij}(s_0, s_1) \wedge s_j(s_1)$.

It consists of the following transitions (the numbers refer to the numbers of the transitions in Fig. 2.3):

(0)–(1)–(2)–(2)–(3)–(1)–(5)–(6)–(7)–(1)–(5)–(9)–(1)–(5)–(8)–(11)–(12)–(13)–(14)–(1)–(5)–(10)–(12)–(15)–(1)–(5)–(10)–(12)–(16)–(1)–(5)–(10)–(12)–(17)

The principle of enforcing an execution of the implementation that corresponds to the execution of a sequence of transition by passing the calculated input data to it is called "guided simulation" (by transitions).

4.3



Testing Single Transitions

In the previous section, we described how to test a system using a transition sequence for one of the automata it contains. Another relevant test scenario is to choose just *one* single transition in an automaton of the specified system and to verify if the implementation behaves correctly with respect to this transition.

For this purpose, we have to find a test sequence that at some point contains the execution of a particular transition $trans_j$. Due to our approach, we have to restrict the search to test sequences of a particular length $k + 1$. Such a test sequence is then specified using a similar predicate P as above in the determination of input data for transition sequences, only this time we only have one transition $trans_j$ and connect the corresponding terms with "$\vee$" (as $trans_j$ need only be executed in one of the steps):

$P = T_{trans_j}(s_0, s_1) \vee T_{trans_j}(s_1, s_2) \vee \ldots \vee T_{trans_j}(s_{k-1}, s_k)$

(see Fig. 4.10 for an illustration).

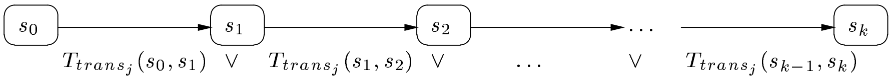


Figure 4.10: Specifying Execution of a Single Transition

Now we again have to solve the formula $M \wedge P$ to determine the desired execution sequence and the corresponding input data. If we do not find a solution, there is no test sequence of length $\leq k + 1$ containing the execution of transition $trans_j$. If we find a solution, the result are particular values for the state variables of the states $s_0, s_1, \ldots, s_k$. By verifying for which i the term $T_{trans_j}(s_i, s_{i+1})$ evaluates to `True` we can determine at which step the transition $trans_j$ is actually executed.

Unfortunately, it is not possible to simplify $M \wedge P$ as we could do in case of the transition sequences, because we do not know in advance at which step $trans_j$ is executed. Therefore, finding a test sequence containing the execution of a particular transition $trans_j$ is much more complex than determining input data for transition sequences.

4.3.1 System Testing Using Separate Transition Test

The main application of this test scenario is to test a system by separately testing its transitions. For this purpose, we choose transitions $trans_j$ of the system (either all transitions or a subset of the transitions we are particularly interested in). Then for each transition, we calculate test sequences containing that transition. Each test sequence shows us an example of what has to happen inside the system so the transition can be executed, and how the system behaves when executing the transition. Thus, we can verify the implementation with respect to one possible behaviour of the system where this transition is involved.

Particularly interesting transitions could be:

- Transitions corresponding to the **termination of a particular task** of the system. An example would be the transition (14) in the `Till` component from state `WaitForAnswer` to state `Ready` that contains the output "`Display!GoodBuy`". The execution of this transition signifies that the customer has successfully withdrawn a certain amount of money. The corresponding test sequence would give us exemplary input data for this case we can use to verify the implementation.

- Transitions corresponding to **error cases** (e.g. the banking transaction is aborted because of a timeout, transitions (3) or (7)). Some error cases could be very rare and it could be difficult to manually find input data. With test sequence determination using SATO, this can be done automatically.

- Transitions corresponding to **communication points** (i.e., a value is read from an input port). With test sequence for such transitions, we can check if the component behaves correctly when communicating with other components. In the `Till` component, every transition reads values from the input ports, but in general, systems also do internal calculations without interfacing with the environment. This is especially the case if we consider larger systems consisting of a hierarchy of multiple components and we are only interested in the communication between the components at a certain hierarchy level (and not in the internal communication between sub-components). For example, in the cash point system (see Fig. 2.1) we might want to verify the communication between the `Centrale` and the `Connections`, so we do not need to consider transitions that pass on data internally within the components the `Centrale` consists of.

We could also calculate *all* test sequences of maximum length $k + 1$ containing to the execution of a particular transition $trans_j$ (instead of just an example). This way, it is possible to verify the correctness of a transition in all possible system executions (of that maximum length) in which it occurs. This can be used as an alternative to the "Separate Function and Trace Test" (SFTT) strategy given in [SK98] that is also based on testing single transitions.

In the SFTT method, first the transitions are tested separately and then the structure of the automaton is verified without considering the actual behaviour of the transitions anymore. Thus, the SFTT method requires that the transitions can actually be tested separately (i.e. we have separate, testable implementations of the transitions). In addition, to be able to verify the structure of the automaton, each transition must always be executable independently of the values of the local variables (*feasibility*) and we must be able to recognize the execution of the transition by its output (*output dististinguishability*). If this is not the case, the system must be modified accordingly, if it is possible.

With our method, we can test the transitions within the context of the whole system, so we do not need additional requirements. However, the system has to be deterministic. Moreover, the number of all test sequences containing the execution of a particular transition will most probably be very large — even though we restricted our verification to test sequences of a limited length.

4.3.2 Checking Executability of Transitions

We can also use the determination of test sequences containing the execution of a particular transition to check if this transition can be executed at all. There are three possible reasons why we might not find a test sequence for a transition:

- The specification is incorrect, there should be cases where this transition is executable. To find out the exact problem, we could try to determine test sequences that put the system into the source state of the transition (see the following section) and check why exactly the transition cannot be executed.
- We found a transition that is unnecessary and can safely be omitted from the specification.
- We chose a too small value for k — the transition is only executable at a later step. If we want to check if a particular transition can really never be executed, we have to choose k high enough so it covers all possible executions of the system (k is then the so-called *diameter* of the system, see [BCCZ99]).

4.3.3 Continuous Testing of Transitions

To test transitions of the system as shown above, we always have to put the implementation into its initial state before applying a test sequence. This might not be possible or not desired. For example, for some transitions, a fairly long test sequence is needed to verify its execution if we start from the initial state.

With a little modification, we can also immediately test other transitions after we tested a particular transition $trans_j$. For this purpose, we simply have to replace the condition $I(s_0)$ for the initial state in the formula M with a specification for the global state we just reached after executing $trans_j$. We know this global state, because it is part of the execution sequence we calculated.

This method can be used to manually test a number of transitions in a row, without having to reset the system. The order of the transitions must be chosen carefully, because not all transitions might be executable anymore after the execution of other transitions. However, many systems in practice have a cyclic structure, so this is mostly not a critical problem. For example, in the `Till` component, the state `Ready` can always be reached again, and from there, any transition can be executed at some point in time. If a particular transition is not executable from the global state reached (in the next $k+1$ steps), the tester could interactively choose other transitions and check if they are executable.

For an illustration of this test method, see Fig. 4.11.

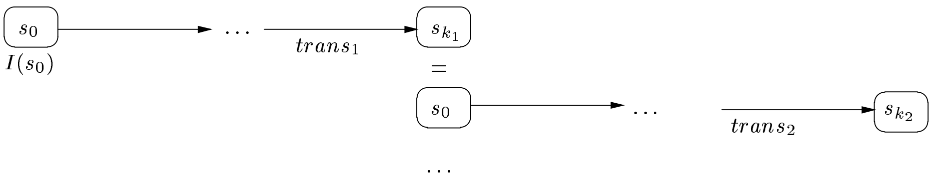
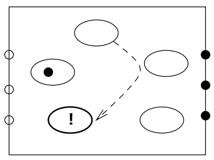


Figure 4.11: Continuous Testing of Transitions

4.4  Putting an Automaton into a Specified State

In a similar way to determining test sequences that contain the execution of a particular transition, we can also determine test sequences that put the automaton of some component

$comp_x$ into a certain control state $state_j$. For this purpose, we specify with P that the control state of that automaton must be $state_j$ in one of the execution steps. Remember we denote the control state of the automaton of component $comp_x$ in step i as $comp_x\_state.i$, so the corresponding predicate P is

$$P = (comp_x\_state.0 == state_j) \vee (comp_x\_state.1 == state_j) \vee \ldots \vee \\ (comp_x\_state.k == state_j)$$

(see Fig. 4.12 for an illustration).

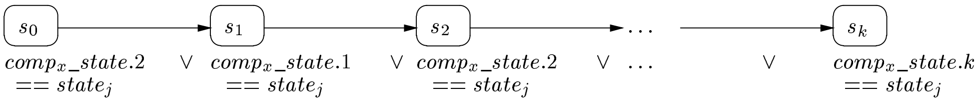


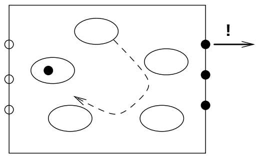
Figure 4.12: Specifying that a Certain State is Reached

Such test sequences can then be used to find out what input data we need so that a certain automaton in the system reaches a specified control state and to check if the corresponding behaviour of the implementation is correct. If the behaviour is correct and the system is deterministic, we then know that the the automaton is in a particular control state (or more exactly, the implementation has reached a point in execution that corresponds to the automaton in the specification being in this control state). Then we can test the behaviour of the implementation if it is in this state.

Interesting control states of the automata we might want to test are control states corresponding to the termination of a particular task (e.g. the state `Ready` is reached again in the `Till` component) or error states, similar to the testing of single transitions (see section 4.3.1).

Like for the transitions, we can also check if a certain control state of an automaton is reachable at all using a test sequence of maximum length $k + 1$. The possible reasons are analogous to the reasons why a particular transition might not be executable (see section 4.3.2).

4.5



Specifying an Output Message

Another interesting test scenario we want to mention is specifying output messages. That means that we want to determine input data so that at one of the steps of the execution, a particular data value is output at a certain output port. This is done in exactly the same way as described in section 4.4, only that we now require the variable for an output port to have a particular value in one of the steps, rather than the variable for the control state of an automaton.

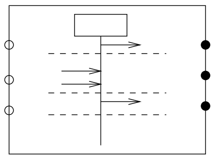
Specifying an output message can be used as an alternative to specifying the execution of a single transition or putting the automaton of a component in a particular control state. It is another way to determine input data to test a certain behaviour of the system.

For example, for the `Till` component, we could specify

- `Display!Timeout` to find input data that leads to a timeout of the system,

- `Display!NoMoneyAvailable` to find input data so that the system reports that no money is available,
- `Money!500` to find input data so that at the end of the execution, the costumer gets 500 credits,
- etc.

4.6



Testing EETs

We already explained that extended event traces (EET's, see section 2.6) can be used to visualize a calculated test sequence by showing the data we have to pass to the input ports at the different steps of the execution and the output data we expect to receive as a response.

Conversely, we can also **use EETs to specify test cases**: an EET specifies a communication behaviour of a system by showing which messages are sent and received at the ports in a possible execution of the system. EETs do not necessarily have to specify the input and output data at every port of the system at each step. There are different possible semantics for EETs — for our purposes, we assume if a port is left out at some step, it can send or receive anything, including the empty message `NoVal`. EET can also contain free variables (e.g., `i?`x denotes that at the port `i` we receive an arbitrary value x). We can now use the logical model of the system specification to determine the complete input and output data for such a test case, in particular also possible values for the free variables and the transitions that are executed in the specification automata to lead to such a behaviour. If we cannot find input data corresponding to a particular EET, we know that it is incompatible to the specification, so we found an error — either in the EET or in the specification.

4.6.1 Verifying Use Cases

We can use the determination of input data for EETs to verify *use cases*. A use case is an EET that describes a required behaviour of the system. Use cases are normally created at the very beginning of system development, before there is a formal system specification or an implementation.

When we later have created a specification and an implementation, we can do the following:

- if it is impossible to determine input data for an EET describing a use case from the logical model of the specification, then the specification and the use case are incompatible. Thus, either there is an error in the specification, or the use case is not valid anymore and should be removed from the system description.
- if we can find input data, we can simulate and verify the use case in the implementation by passing the calculated input data to it and checking if the output data is as expected
- the sequence of transitions that is executed in the specification automata for a behaviour corresponding to a certain use case gives us interesting additional information and insight into the system

4.6.2 Example

The EET in Fig. 4.13 shows a use case for the `Till` component where the customer withdraws money: first the card is inserted, and at a later step the customer enters the command to withdraw the desired amount (500 credits). Finally, the money is given to the customer and the display shows the message "Good Buy". Most input and output values of this test case are left open, we are just interested in the facts we mentioned (insertion of the card and successful withdrawal of the correct amount of money).

A test sequence for this scenario is shown in Fig. 4.14. This test sequence was determined automatically and shows the complete input and output data necessary to simulate the behaviour we specified by the EET in Fig. 4.13.

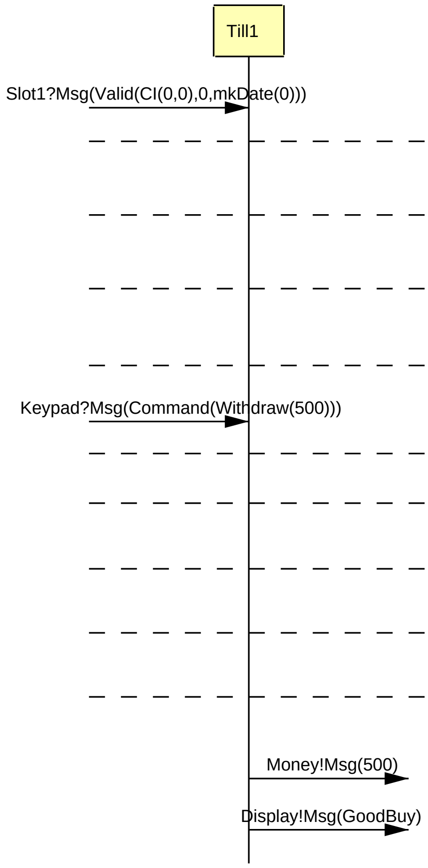


Figure 4.13: EET for a Test Case

4.6.3 How to Determine Test Sequences for EETs

As a simplification, we only consider EETs describing the communication of *one* component. We also assume the EET does not have a hierarchical structure or uses concepts like alternatives or loops (i.e. sub-EETs inside the EET that can be executed one or more times as part of the execution of the whole EET).

With this simplification, it is easy to translate an EET into the predicate P we need to specify a test case: the EET consists of a number of execution steps separated by the tick marks (dashed lines). These steps correspond to the steps $s_0, s_1, \ldots$ of an execution sequence in the logical system model. In the sections between the tick marks, the EET

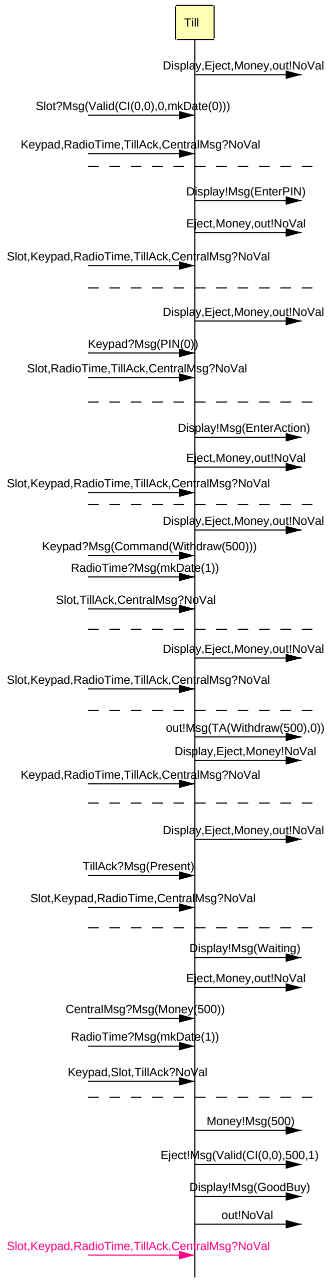


Figure 4.14: Test Sequence for the EET in Fig. 4.13

specifies data values for the ports. Thus, if we have a message `port?expr` at step i in the EET, we have to add the term `port.i==expr` to the predicate P, meaning that at step i the value given by `expr` must be input at the port `port`.

For example, the test case shown by the EET in 4.13 can be translated to

$$\begin{aligned} P = {} & (\mathbf{Slot}.0 == \mathbf{Msg}(\mathbf{Valid}(\mathbf{CI}(\mathbf{0},\mathbf{0}),\mathbf{0},\mathbf{mkDate}(\mathbf{0}))))\wedge \\ & (\mathbf{Keypad}.4 == \mathbf{Msg}(\mathbf{Command}(\mathbf{Withdraw}(\mathbf{500}))))\wedge \\ & (\mathbf{Money}.9 == \mathbf{Msg}(\mathbf{500})) \wedge (\mathbf{Display}.9 == \mathbf{Msg}(\mathbf{GoodBuy})). \end{aligned}$$

Then, we calculate a solution for $M \wedge P$ as usual and determine the input and output data for the execution steps of the test sequence. From the solution, we can also determine the sequence of transitions to be executed in the specification automata to lead to that behaviour, by inserting the actual values for the state variables at the different execution steps into the terms $T_{trans_j}(s_i, s_{i+1})$ for the transitions and checking for which transition the corresponding term evaluates to `True`.

In the same way, it is also possible to translate EETs describing the communication of more than one component at the same time into a predicate P. Also, hierarchical EETs could be translated, though only in a limited way. We will leave the examination of this topic as further work.

4.6.4 EETs Containing Free Variables

EETs can also contain free variables. Imagine in Fig. 4.13 we had written "x" instead of "500" both times it occurs. This use case then specifies that the customer receives exactly the same amount of money as demanded — either in all cases or in a particular case, depending on if we consider x as all-quantified or as existentially quantified. If we replace "500" by "x" the first time and by "$x+1$" the second time, the customer receives one credit more than demanded, which should not be possible.

Existentially Qualified Variables

Fortunately, existentially qualified variables are no problem for our determination of input data. We just keep them as additional free variables when we translate the EET into a predicate P. The solution for the formula $M \wedge P$ that gives us an execution sequence of the system corresponding to the EET then also contains the right values for the free variables so that we have a valid execution of the system.

All-Quantified Variables

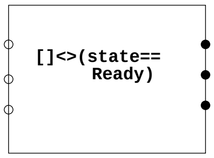
If the variables in the EET are all-quantified (i.e., the behaviour in the EET should be valid for all possible values for these variables), we have to modify the translation slightly: the second and all following times such a variable occurs in the EET, we negate the equation. For example, if we replace "500" by x in the EET in Fig. 4.13, we have:

$$
\begin{aligned}
P = {} & (\mathbf{Slot}.0 == \mathbf{Msg}(\mathbf{Valid}(\mathbf{CI}(\mathbf{0},\mathbf{0}),\mathbf{0},\mathbf{mkDate}(\mathbf{0}))))\wedge \\
& (\mathbf{Keypad}.4 == \mathbf{Msg}(\mathbf{Command}(\mathbf{Withdraw}(\mathbf{x}))))\wedge \\
& (\mathbf{Display}.9 == \mathbf{Msg}(\mathbf{GoodBuy}))\wedge \\
& \mathbf{not}(\mathbf{Money}.9 == \mathbf{Msg}(\mathbf{x}))
\end{aligned}
$$

Now, if there is no solution for $M \wedge P$, the behaviour described in the EET is possible for all values of x. On the other hand, if we find a solution, we have a *counter example*: an execution where a particular amount x of money is demanded but a different amount (or nothing at all) is presented at the output port `Money` together with the message `GoodBuy` on the display.

However, we have to be careful when specifying an execution by an EET, especially if it contains variables — defining an exact semantics for EETs is a difficult topic and many questions in that area are still open.

4.7 Temporal Logic and Bounded Model Checking



The final and most general method we describe to specify test cases is via **formulas in temporal logic**. Unlike classical propositional logic, with formulas in temporal logic we can express *at which points in execution* certain predicates should be satisfied. For example, we can specify that *at some point in future* an automaton reaches a certain state, or that there is an execution where *at all steps* the same value is output at a particular output port. The test scenarios we described in sections 4.4, 4.5 and 4.6 can in fact also be specified by LTL formulas.

Finding input and output data for test cases described by such formulas leads us to the **Bounded Model Checking** algorithm described in [BCCZ99]. Bounded Model Checking can be used as a very efficient alternative to conventional model checking to verify arbitrary properties of a specification, expressed in temporal logic. The algorithms described in [BCCZ99] actually provided the basis and incentive for the specification based test sequence determination methods we explained in this thesis.

4.7.1 Introduction to Temporal Logic

First, we give a short description of linear time temporal logic (LTL). With LTL formulas, it is possible to express properties of execution sequences of a system. However, the execution sequences LTL formulas make statements about are *infinitely long*, i.e. they do not end at a certain step s_k as in our description. A LTL formula f can be:

- A predicate on the state variables of the current step in the execution sequence (e.g. $\mathbf{comp}_x.\mathbf{state} == \mathbf{Ready}$, or $\mathbf{Display} == \mathbf{Msg}(\mathbf{Timeout})$),
- an expression of the form $\mathbf{not}(g)$, $g \wedge h$, $g \vee h$, where g and h are LTL formulas,
- the formula $\circ g$ ("nexttime g"), stating that the LTL formula g is true in the *next* step in the execution sequence,
- the formula $\diamond g$ ("future g"), stating that the LTL formula g is true in the *current or one of the following* steps in the execution sequence,
- the formula $\Box g$ ("always g"), stating that the LTL formula g is true in *all* steps in the execution sequence, starting from the current step.

Fig. 4.15 gives a visual explanation of the temporal operators $\circ$, $\diamond$ and $\Box$.

As LTL formulas only make statements about particular execution sequences, we also have to specify which execution sequences we mean: If f is a LTL formula, then

- $\mathbf{A}f$ states that f is true for all possible execution sequences of the system, and
- $\mathbf{E}f$ states that there is an execution sequence for which f is true.

One can easily see that $\mathbf{A}f$ is the same as $\mathbf{not}(\mathbf{E}\ \mathbf{not}(f))$.

To determine input data for a test case specified by a LTL formula f, we want to find execution sequences satisfying this formula (rather than stating that all executions have to satisfy f), so we are interested in a solution for $\mathbf{E}f$.

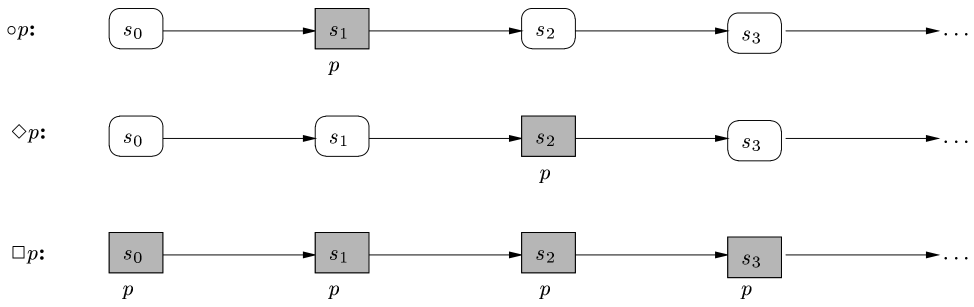


Figure 4.15: Temporal Operators in LTL

Examples

Some examples how to formulate interesting properties in LTL are given in the following (based on the `Till` component of the cash point system):

1. $\mathbf{E}\diamond(\mathbf{Till.state} == \mathbf{Transmitting})$: specifies an execution sequence so that at some point in execution the state `Transmitting` is reached
2. $\mathbf{E}\diamond(\mathbf{not}(\mathbf{Till.state} == \mathbf{Ready}) \wedge \diamond(\mathbf{Till.state} == \mathbf{Ready}))$: the state `Ready` is left and later re-entered (so the system performs a certain transaction which ends somehow — successfully or with an error — and then the system is ready for another transaction).
3. $\mathbf{E}((\mathbf{slot} == \mathbf{Msg}(\mathbf{Valid}(\mathbf{CI}(\mathbf{0},\mathbf{0}),\mathbf{0},\mathbf{mkDate}(\mathbf{0})))) \wedge$ $\circ\circ(\mathbf{Display} == \mathbf{Msg}(\mathbf{ErrorWrongPIN})))$: a valid card is inserted but two execution steps later rejected because the PIN was wrong.
4. $\mathbf{E}(\Box\mathbf{not}((\mathbf{Display} == \mathbf{ErrorTimeout}) \vee (\mathbf{Display} == \mathbf{ErrorWrongPIN}) \vee (\mathbf{Display} == \mathbf{ErrorDayLimitExceeded})))$: an infinitely long execution sequence where no error messages occur.
5. $\mathbf{E}(\Box\diamond(\mathbf{Till.state} == \mathbf{Ready}))$: an execution sequence, so that always the state `Ready` is reached again at some point in future. This means, the state `Ready` is reached infinitely often.

In general, it can be difficult to formulate properties of a system in temporal logic. See [CDA98] for typical specification patterns and how they are translated to temporal logic or to other specification languages.

LTL specification for previously described test scenarios

Note that we can also specify some of the test scenarios we explained in the previous sections in LTL. In sections 4.4 and 4.5 we look for an execution sequence so that at some future step either the control state of the automaton or one of the output ports has a certain value. This corresponds to a LTL formula like given in the first example.

When we specified test cases via EETs (see section 4.6), we specified (part of) the input and output messages received from and sent to the system at certain execution steps. We can make statements about certain execution steps using the LTL "$\circ$" operator, like in the third example. The translation of the EET in Fig. 4.13 would be:

$$\mathbf{E}((\mathbf{slot} == \mathbf{Msg}(\mathbf{Valid}(\mathbf{CI}(\mathbf{0},\mathbf{0}),\mathbf{0},\mathbf{mkDate}(\mathbf{0})))) \wedge$$
$$\circ\circ\circ\circ((\mathbf{Keypad} == \mathbf{Msg}(\mathbf{Command}(\mathbf{Withdraw}(\mathbf{500})))) \wedge$$

$\circ \circ \circ \circ \circ((\mathbf{Money} == \mathbf{Msg(500)}) \wedge (\mathbf{Display} == \mathbf{Msg(GoodBuy)})))))$

We cannot directly translate the test scenarios specifying the execution of transitions (sections 4.2 and 4.3) to LTL, because specifying the execution of transitions always involves two execution steps at once (the current and the successor step).

4.7.2 Test Sequence Determination for LTL Specifications

How do we find test sequences that satisfy a specification we formulated in LTL ? We can also do this via our logical model for AUTOFOCUS specifications as we did before, by solving the formula $M \wedge P$. The only thing we have to do is to show how to translate a LTL formula into a predicate P.

However, there is one problem: as we said, LTL formulas make statements about *infinitely long* execution sequences (e.g., $\Box p$ says that p is true in all steps of an infinitely long execution sequence). On the other hand, our logical model only allows us to specify finite execution sequences, ending at a particular step s_k.

The solution is that finite execution sequences can also act as infinite execution sequences if they contain a **loop** (see Fig. 4.16). If the successor state of the last global state s_k can be one of the previous states (for example s_1), then this execution sequence specifies an infinite behaviour (the execution sequence $s_0, s_1, s_2, \ldots, s_k, s_1, s_2, \ldots, s_k, s_1, \ldots$). An execution sequence contains a loop if $T(s_k, s_i)$ is true for some i.

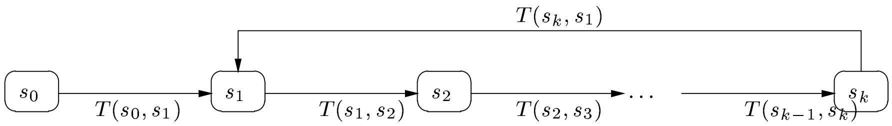


Figure 4.16: Execution Sequence with a Loop

As an example, we consider the `Till` component again. In its initial state s_0, the control state of the automaton is `Ready` and the output ports are cleared. Now imagine the transition sequence (0)–(1)–(5)–(10)–(12)–(14)–(0) is executed. After the execution of transition (0) the message "EnterCard" is output on the `Display`, the corresponding global state is s_1. Now, the other transitions are executed (meaning the customer demands a certain amount of money and receives it). After execution of transition (0) the second time the system is in a global state s_7 which is exactly the same as s_1 (control state `Ready`, message "EnterCard" on the `Display`, other output ports cleared). So from there on, we can execute the transition sequence (1)–(5)–(10)–(12)–(14)–(0) infinitely often. This is an example for an infinite execution sequence where never any error messages occur, i.e. a solution to the fourth LTL example formula given above — even though we only have a finite number of steps.

The other point is that we only consider execution sequences of a maximum length k (whether containing a loop or not) as solutions for a LTL formula f. We call k a **bound**. If we cannot find an execution sequence with maximum length k that satisfies the formula, we cannot say if it is satisfiable or not, because there might be a longer execution sequence that satisfies the formula, or there might be none.

Translation of LTL formulas

We now show how to translate a LTL formula f into a predicate P so that a solution to $M \wedge P$ gives us an execution sequence that satisfies f. However, we will only take the two examples $\Diamond p$ and $\Box p$, where p is a formula not containing any temporal operators anymore. The translation of arbitrary LTL formulas is given in [BCCZ99].

- The translation of the formula $\Diamond p$ with the **future operator** $\Diamond$ is easy: p must be true at one of the steps. Remember that k acts as a bound, so we cannot find any execution sequences where p is true at a later step than k. Thus, if we denote by $p.i$ that p is true at step i,
  $P = p.0 \vee p.1 \vee p.2 \vee \ldots \vee p.k$

- For the translation of the formula $\Box p$ we need an execution sequence containing a loop, as p must be true in all steps of an infinitely long execution sequence. This is exactly how we specify P:
  $P = (p.0 \wedge p.1 \wedge p.2 \wedge \ldots \wedge p.k) \wedge$
  $(T(s_k, s_0) \vee T(s_k, s_1) \vee T(s_k, s_2) \vee \ldots \vee T(s_k, s_k))$.
  The first part specifies that p is true in all steps of the execution sequence, and the second part specifies that it contains a loop.

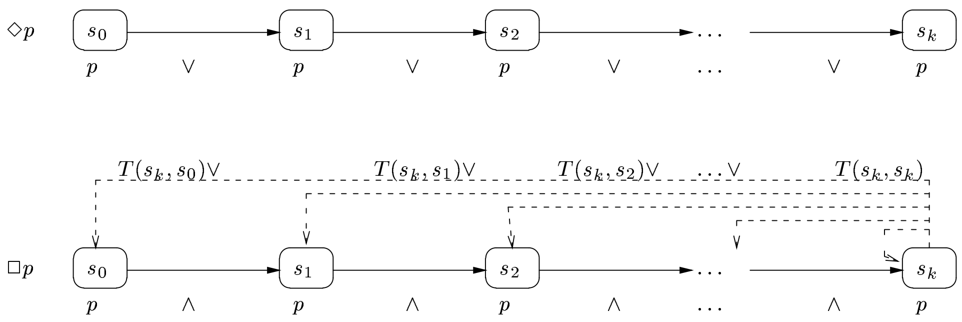


Figure 4.17: Translation of LTL Formulas

Fig. 4.17 shows a graphical visualisation of this translation.

This way, we can now specify system behaviour via LTL formulas and calculate the corresponding test sequence with its input and output data by translating the formula to a predicate P on the execution sequences and calculating a solution for $M \wedge P$.

4.7.3 Bounded Model Checking

In addition to determining input data for test cases described by a LTL formula, we can also use our logical model together with the translation of LTL formulas for *model checking*. In this case, we are not interested in the actual input data, we only want to know if we can determine execution sequences fulfill a certain LTL formula from the specification or not.

This way, we can perform model checking, i.e. formulate desired properties of a system using temporal logic and verify if they are satisfied by the specification.

An important model checking algorithm is symbolic model checking using BDDs, as implemented in the SMV system [McM93]. In section 3.2.6, we have actually shown how to translate AUTOFOCUS specifications to the SMV input language, so we can use SMV as a model checker to verify these specifications.

However, the approaches described in this thesis provide an efficient alternative: ***bounded model checking*** [BCCZ99]. Bounded model checking can be used in the same way as conventional model checking, but it only considers executions of a maximum length k ("bound"). For example, if we get the result that $\mathbf{E}\Diamond p$ is not satisfied for execution sequences of maximum length k, this does not necessarily mean that it cannot be satisfied at all: it might be that we just chose a too small value for k. To be on the safe side, we would have to choose a k so that it corresponds to the longest possible execution sequence of the system (the so-called "diameter"). Unfortunately, such a value for k is usually far too large.

Now, bounded model checking works as follows:

- If we have the property $\mathbf{E}f$ (there **exists an execution** where the LTL formula f is true), we try to find an execution sequence of maximum length k fulfilling the formula f. We can do that as we explained above, by translating f to a predicate P and solving $M \wedge P$. If we find a solution (no matter what the actual input data is), the specification fulfills $\mathbf{E}f$, otherwise it does not. We only have to keep in mind that we only consider execution sequences of maximum length k.
- If we have the property $\mathbf{A}f$, f should be fulfilled **for all possible executions**. Thus, we try to find an execution sequence that fulfills $\mathbf{not}(f)$. If we do not find such an execution sequence, the specification fulfills $\mathbf{A}f$, otherwise the calculated execution sequence gives us a *counter example* showing that f is not true for all executions of the system. Because of the bound k, we only get counter examples of a maximum length k as a result — so if we do not find a solution we cannot be sure that $\mathbf{A}f$ is really satisfied, because there might be longer counter examples.

As part of our work, we implemented the translation of arbitrary LTL formulas to predicates P (for which section 4.7.2 only shows two special cases). This way, now a **bounded model checker for** AUTOFOCUS is available, that can be used instead of SMV if only counter examples of a maximum length k should be considered. For a small k, bounded model checking is far more efficient than conventional model checking — we will show an example for that in the evaluation (see chapter 6).

4.8 Testing Non-Deterministic Systems

This concludes our explanations of ways to determine test sequences together with input and output data for particular test scenarios from an AUTOFOCUS specification. We finish this chapter with some interesting thoughts about testing non-deterministic systems.

As explained in section 4.1.3, if we have a non-deterministic system it is possible that we calculate a test sequence we cannot use to verify the implementation by passing the input data to it and checking if the output data is as expected. At some point in execution, the implementation might have a "choice" between more than one possible behaviour, and it might choose the "wrong" alternative (e.g. not the execution specified by the test sequence). Thus, it is generally not possible to force the system to behave in the required way.

In this section, we present some ideas to tackle this problem. However, the descriptions will be fairly short and we leave most of this topic as further work.

4.8.1 Detecting Non-Deterministic Test Sequences

If the input data given by a test sequence can lead to more than one possible behaviour of the system, we call it a *non-deterministic test sequence*. As the transitions in the automata are deterministic (they write fixed values to the output ports and local variables), a test sequence is non-deterministic if there is a point in execution where two transitions are enabled at the same time – we can detect this by checking if more than one $T_{trans_j}(s_i, s_i + 1)$ evaluates to `True` in the calculated execution sequence.

This way, we can output a warning that the behaviour of the implementation might differ from the behaviour specified by this test sequence.

4.8.2 Calculating All Solutions

Another possibility is to modify the Davis-Putnam procedure so that all solutions for the logical formula specifying the test sequence are calculated, instead of just one (the program SEMPROP [Let99] has this feature and could be used instead of SATO). Then we can see if one of the solutions is deterministic. However, this modification makes the Davis-Putnam procedure much more complex and slower.

4.8.3 Pseudo-Deterministic Test Sequences

If we are lucky, non-determinism might not influence the desired property of our test sequence — for example, if we want to find a test sequence corresponding to reaching a certain control state in the specification, this might be possible independent of non-deterministic choice at some point (imagine the transitions in Fig. 4.6 would both point to a second state, then this state is always reached in the next step, in spite of the non-determinism).

We can detect such test sequences as follows: every time we encounter a non-deterministic choice between two or more transitions (so instead of the transition chosen in this test sequence, other transitions could have been executed), we calculate additional test sequences where we specify that the other transitions must be executed and the predicate P must still be satisfied. If this is possible, we can enforce the desired behaviour of the system even though it is non-deterministic.

Unfortunately, now we need complete insight into the global system state, so that we know which of the test sequences reflects the actual behaviour of the system. If this is not possible, we additionally have to require that all these test sequences have the same input data, which makes finding them much more difficult.

4.8.4 Specifying Deterministic Test Sequences

Another way is to change the transition relation we derive from the AUTOFOCUS system model such that a transition in an automaton can only be executed if it is the only enabled transition. Then we automatically find non-deterministic test sequences. However, this makes the transition relation much more complex, and we will not find pseudo-deterministic or non-deterministic transition sequences anymore.

4.9 Conclusion

In this chapter, we showed how to specify test cases for systems modelled with AUTOFOCUS and how to calculate test sequences containing the necessary input data and output data to be passed to and to be expected from an implementation in this case.

Such test sequences are determined by formulating a predicate P on the steps of the system execution and then using the logical system model described in Chapter 3 to find an execution satisfying P by solving the formula $M \wedge P$.

In particular, we described six different test scenarios: determining input data for transition sequences, testing single transitions, putting an automaton into a specified state, specifying output data, testing EETs and specifying test sequences via logical formulas.

The last test scenario lead us to a bounded model checking algorithm for AUTOFOCUS systems, making it possible to verify in an efficient way if arbitrary properties expressed in temporal logic are fulfilled for a particular system specification.

Finally, we mentioned some ideas how to solve the problem of testing non-deterministic systems.

This concludes our explanation of specification based test sequence determination. We will continue with a description of our implementation and a performance evaluation.

Chapter 5

Implementation

As part of our work, we have implemented a framework for specification based test sequence determination in the programming language Java. Our implementation is an extension to the QUEST tools for AUTOFOCUS (see [BLS99]), which provide a Java class model for the various system description techniques (such as SSD's or STD's, see Chapter 2) and functionality for simulation, verification and code generation for AUTOFOCUS system models. The user can access this functionality via the AUTOFOCUS model browser.

We added classes to the QUEST project that make it possible to translate AUTOFOCUS specifications to propositional logic as described in Chapter 3, which provides the basis for arbitrary specification based test data determination. Calculated execution sequences representing the test data ("counterexamples") can be translated to MSCs and can be displayed by the browser. In addition, a fully functional implementation exists of the test data determinination for transition sequences (see section 4.2) and for single transitions (section 4.3), of the bounded model checking algorithm (section 4.7) and of the SMV translation (section 3.2.6). The implementation of the other test scenarios described in chapter 4 (especially testing EETs, see section 4.6) has been left as further work.

Our implementation consists of 11 Java classes, which altogether contain approximately 8000 lines of source code.

In this chapter, we describe the structure and usage of our implementation. More detailed information can be found in the HTML documentation generated from the commented source code via JavaDoc. For a general introduction to the usage of the QUEST tools, the reader is referred to the QUEST manual [BLS99]. Performance statistics obtained using our implementation are presented in the following chapter.

5.1 Class Structure of the Implementation

The classes of the implementation are depicted in Fig. 5.1. The lower part corresponds to translation of AUTOFOCUS specifications to propositional formulas by the translation steps described in Chapter 3 — in general there is a Java class for each major translation step. These translation functionalities are then used by classes implementing the test data determination for the different test scenarios (see Chapter 4, e.g. input data determination for transition sequences or bounded model checking) or the export to SMV.

The implementation consists of the following classes, which together form the Java package `quest.sato`:

- The class **`LogicalComponentModel`**, which represents the operational semantics of an AUTOFOCUS component and contains methods to determine its state space, initial states and transition relation (see section 3.1).

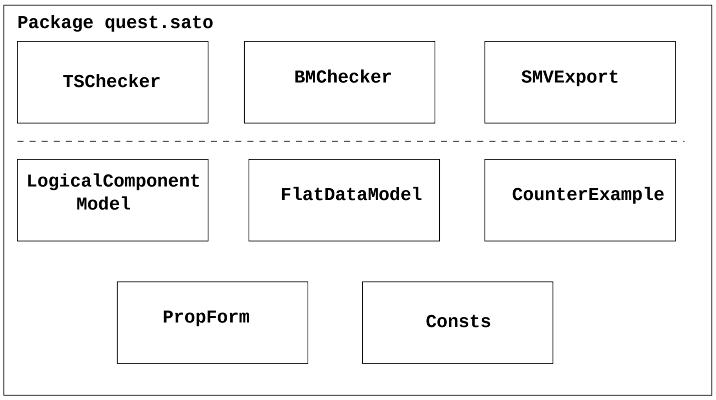


Figure 5.1: Class Structure of the Implementation

- The class **FlatDataModel** (together with two subclasses and an exception class), which offers functionality to eliminate user-defined data types and functions (see 3.2).
- The class **PropForm**, which provides utility functions to create and transform boolean valued terms, to translate boolean valued terms into propositional formulas (see section 3.3) and to solve propositional formulas using SATO (see section 3.4).
- The class **Consts**, which defines constants for frequently used terms and operators (such as $\vee$, $\wedge$ etc.).
- The class **Counterexample**, which makes it possible to translate a solution to a propositional formula specifying a test case back to an AUTOFOCUS MSC that shows the input and output data calculated for this test case.
- The classes **TSChecker**, **BMChecker** and **SMVExport**, which make it possible to determine test data for transition sequences and single transitions, or offer bounded model checking respectively the export to SMV. The functionality of these classes can be invoked from the AUTOFOCUS model browser, other classes in the QUEST project, or from the command line.

Fig. 5.2 shows a message sequence chart that illustrates how the class `TSChecker` cooperates with the classes `LogicalComponentModel`, `FlatDataModel`, `PropForm` and `Counterexample` to determine test data for a transition sequence (see section 4.2): the formula $M \wedge P$ is built by combining the terms for initial states and transition relation (obtained using the class `LogicalComponentModel`) and an additional condition ensuring that the variables always have legal values (see section 3.2.4). At the same time, these terms are translated to propositional formulas using `FlatDataModel` and `PropForm`. Finally, SATO is called (via the class `PropForm`) to solve the formula $M \wedge P$, and the result is translated back to a MSC using `Counterexample`.

In the following, we will briefly describe the single classes of the implementation, their usage and special features.

5.2 The Class **LogicalComponentModel**

Fig. 5.3 shows the class `LogicalComponentModel`, together with its relationships to other classes, its constructor and its most important methods. With the help of this class, it is possible to create a logical model for an AUTOFOCUS component that represents

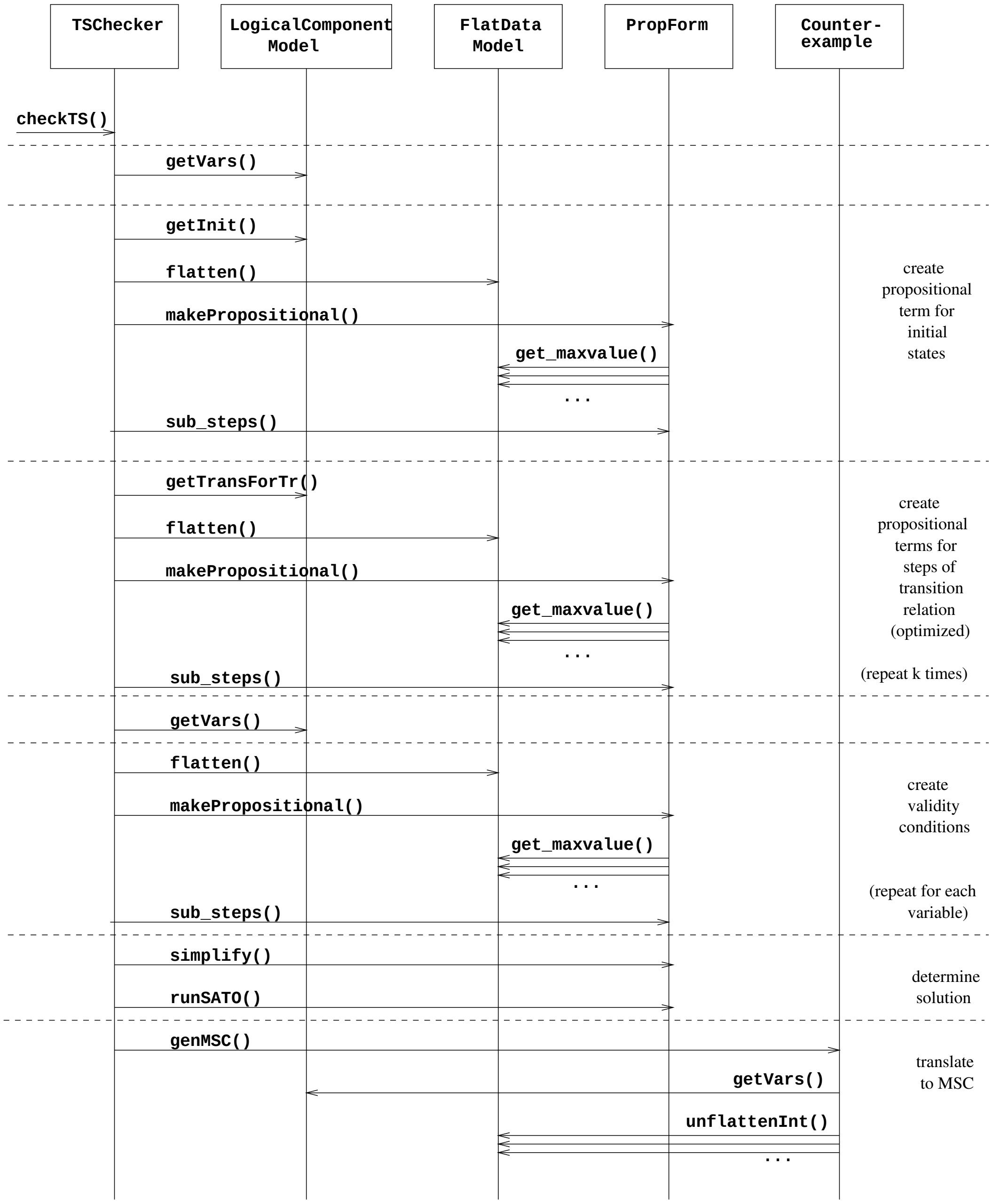


Figure 5.2: Determining Test Data for a Transition Sequence

its semantics as a state transition graph, as described in section 3.1. For this purpose, it contains methods to determine the state variables of the model, and to determine the initial states and the transition relation as boolean valued terms. For those terms, the class `Term` defined in the QUEST framework is used.

A logical model for an AUTOFOCUS component is created via the **constructor** of `LogicalComponentModel`, which takes the corresponding component and a data type definition (`DTD`) as arguments. The data type definition is needed so that the new data type for the control states of the automata can be added to it (see section 3.1.2). The component and the data type definition the logical component model is based on can then be retrieved

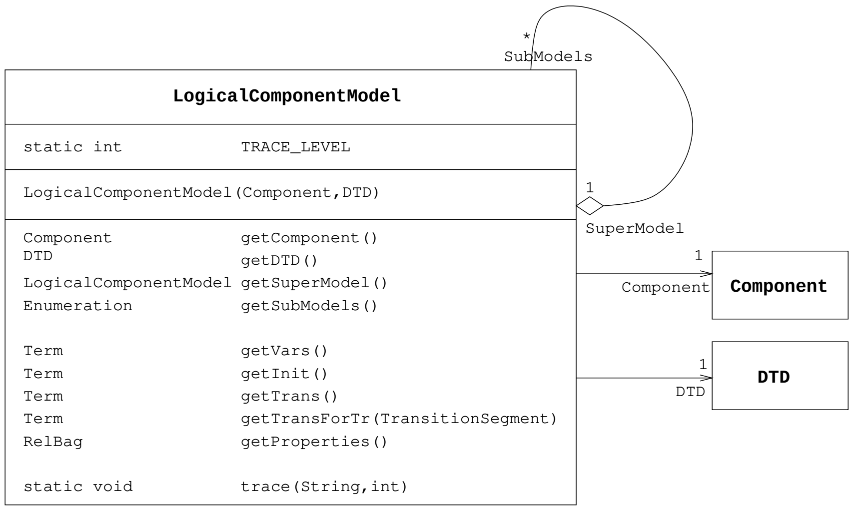


Figure 5.3: The Class `LogicalComponentModel`

via the methods **`getComponent()`** and **`getDTD()`**.

If an AUTOFOCUS component contains subcomponents, logical models (i.e. the state variables, initial states and transition relation) are also determined for all the subcomponents. They can be accessed using the method **`getSubModels()`**. Conversely, the method **`getSuperModel()`** returns the logical component model for the super component.

With the other methods, one can determine the state variables, initial states and transition relation, which give the semantics for the component:

- **`getVars()`** returns an enumeration of the state variables
- **`getInit()`** returns a boolean valued term containing these variables that specifies the initial states
- **`getTrans()`** returns a boolean valued term specifying the transition relation
- **`getTransForTr(TransitionSegment ts)`** returns a term specifying the transition relation, with the additional requirement that the transition `ts` must be executed. This is needed to determine input data for transition sequences and corresponds to the term $(T(s_i, s_{i+1}))_{trans_i}$ explained in section 4.2.
- For model checking purposes, one can specify properties for AUTOFOCUS components as formulas in temporal logic. The method **`getProperties()`** returns an enumeration of the properties specified for the component and its subcomponents. Possible pattern matching on ports in the property terms is eliminated.

Trace functionality

Finally, the class `LogicalComponentModel` contains a trace functionality to facilitate testing and finding errors in the implementation. For this purpose, there is a static variable `TRACE_LEVEL` which can be set by other classes. The methods in `LogicalComponentModel` report various information on their progress via calls to the static method `trace`. The higher the trace level is set, the more of this information is output to the console. With a trace level of 0, nothing is output at all.

This trace functionality is also contained in all other classes in the package `quest.sato` and works in the same way. Therefore, we will not repeat its description in the following sections.

5.3 The Class **`FlatDataModel`**

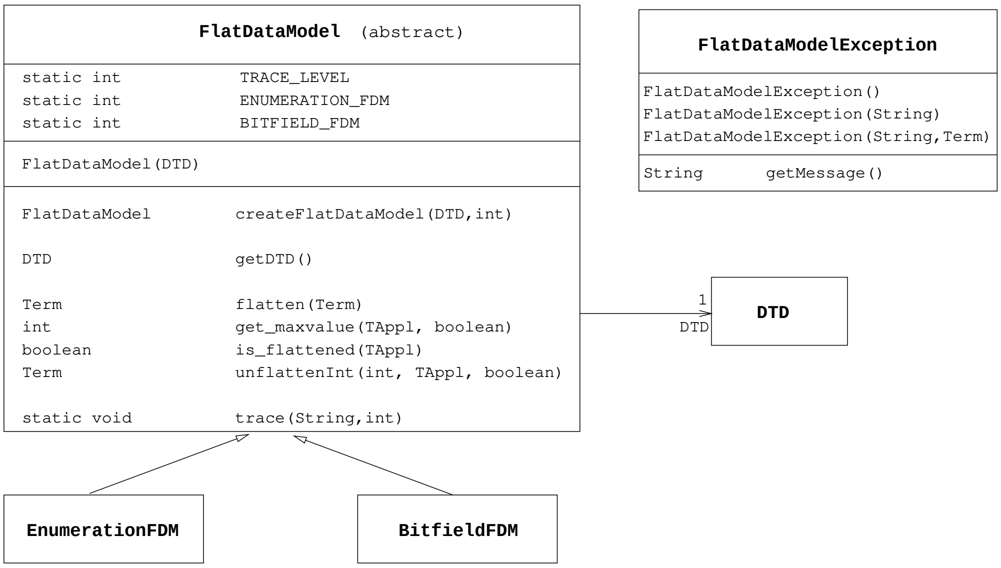


Figure 5.4: The Class `FlatDataModel`

The class `FlatDataModel` (see Fig. 5.4) is responsible for the elimination of the user-defined data types and functions from AUTOFOCUS terms, as described in section 3.2.

In this section, we also pointed out that there are different possibilities to map AUTOFOCUS data types to integer values. The class `FlatDataModel` is therefore an abstract class, containing the generic functionality needed to eliminate user-defined data types and functions. The different possibilities to perform this elimination (by using different mappings) are then represented by subclasses of `FlatDataModel`. In particular, we implemented the two methods described in section 3.2 — the corresponding classes are `EnumerationFDM` (for the enumeration method) and `BitfieldFDM` (for the bitfield method).

To be able to make use of the methods to eliminate user-defined data types and functions, first an instance of `FlatDataModel` (or more specifically, of one of its subclasses) must be created. This is done via the **constructor** `FDM(DTD)`, which takes an AUTOFOCUS data type definition as an argument. The integer mapping of the data types contained in this data type definition and the replacement terms for functions, constructor, discriminator and selector terms are then pre-calculated. For convenience, there is a static method `createFlatDataModel(DTD dtd,int type)` in `FlatDataModel` which saves the programmer from calling the constructors of the subclasses. It returns either an instance of `BitfieldFDM` or of `EnumerationFDM` depending on if its second argument is `FlatDataModel.BITFIELD_FDM` or `FlatDataModel.ENUMERATION_FDM`.

Now, the following methods can be used:

- **`getDTD()`** returns the data type definition the `FlatDataModel` is based on.

- **`flatten(Term t)`** eliminates the user-defined data types and functions from the AUTOFOCUS term `t` as described in section 3.2, and returns the result.
- **`get_maxvalue(TAppl type, boolean isChannel)`** returns the maximum integer value mapped to a value of the data type `type`. The second argument `isChannel` indicates if we want to know the maximum value of `type` itself, or of a channel or port of this type.
- **`is_flattened(TAppl type)`** returns `True` if an integer mapping for the data type `type` is available within this `FlatDataModel` object, `False` otherwise.
- **`unflattenInt(int value, TAppl type, boolean isChannel)`** reverses the mapping of AUTOFOCUS data types to integer values. It calculates the constructor term of type `type` mapped to the integer value `value`. `isChannel` again indicates whether or not we refer to a channel or port of type `type`.

Error Handling

It is possible that a `FlatDataModel` object cannot be created for a certain data type definition, for example if not all defined data types are finite. In this case, the constructor (and therefore also the method `createFlatDataModel`) throws a `FlatDataModelException` containing a corresponding error message (to be obtained via `getMessage()`). Such an exception is also generated by the method `flatten()` in case the term given as an argument contains data types not defined in the data type definition given to the constructor or if there are other errors.

5.4 The Class `PropForm`

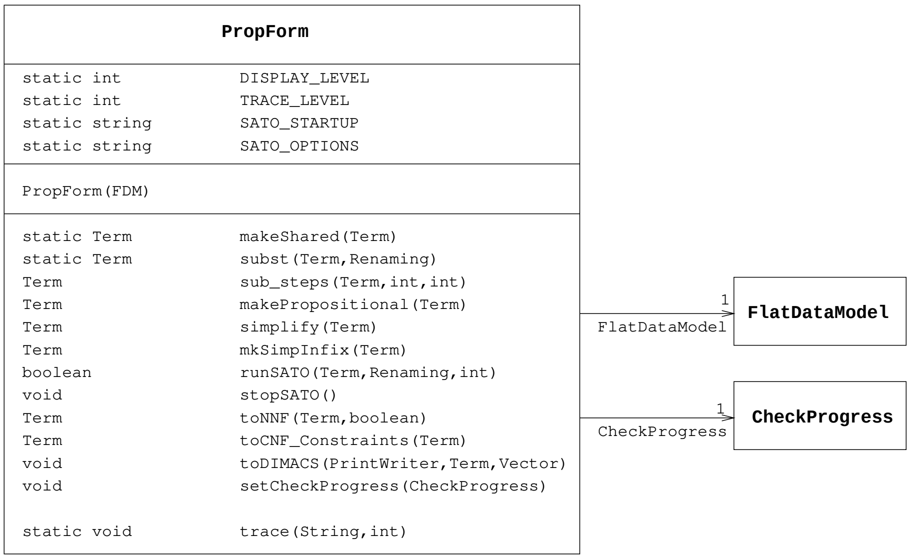


Figure 5.5: The Class `PropForm`

The class `PropForm` (see Fig. 5.5) contains methods

- to create and transform boolean valued terms, with support of shared subterms

- to translate boolean valued terms to propositional formulas, and
- to calculate a solution for a propositional formula using SATO

Utility Methods and Shared Subterms

The utility methods to create and transform boolean valued terms will not be described here in detail, most of them just offer additional functionality not found in the QUEST class `Term`. For example, the method **`sub_steps()`** substitutes the state variables in a term so that they refer to a particular execution step (e.g. to determine $T(s_i, s_{i+1})$ from $T(s, s')$).

However, an important feature of `PropForm` is the handling of terms containing ***shared subterms***. Usually, a term (like $(a \wedge b) \vee (\mathbf{not}(a \wedge b) \vee c)$ is stored as a tree. The method **`makeShared`** scans such a term for common subterms (in this case, $(a \wedge b)$) and replaces them by *one single* `Term` object. Thus, the term is now stored as a directed acyclic graph (DAG) rather than a tree.

In our translations of AUTOFOCUS models to boolean valued terms, often subterms occur more than once, so this approach saves considerable memory. In addition, it also speeds up the translations (as common subterms only have to be translated once) and in fact even reduces the number of clauses in the resulting conjunctive normal form. The latter is due to the fact that for a shared subterm only one constraint has to be generated when determining the CNF (see section 3.4.3). However, now all methods working on terms have to make sure the sharing is preserved, so `PropForm` for example contains a special version of **`subst`** to substitute variables in terms, which replaces the `subst` method of the `Term` class.

Constructor

The utility methods in `PropForm` are static. To be able to use the other methods, an instance of `PropForm` has to be created with the **constructor** `PropForm(FDM)`, which takes a `FlatDataModel` object as an argument.

Translation to Propositional Formulas

With the help of this `FlatDataModel` object giving the integer mapping for the data types, a boolean valued term can be transformed into a formula in propositional logic by separating the variables into their bits (see section 3.3). This is done via the method **`make-Propositional`**.

Calculating a Solution for a Propositional Formula

The other methods in `PropForm` make it possible to calculate a solution for such a propositional term, as described in section 3.4:

- **`simplify(Term t)`** simplifies a propositional term as shown in section 3.4.1. `PropForm` also provides methods to simplify a term during its creation. For example, **`mkSimpInfix(Term a, Term op, Term b)`** creates an infix term `a op b` which is immediately simplified. Thus, `mkSimpInfix`$(x, \wedge, \mathbf{True})$ would return just x.
- **`toNNF(Term t, boolean eliminateEquivs)`** converts a propositional term `t` into negation normal form (see section 3.4.2). The second argument indicates if during the conversion the operator $\Leftrightarrow$ should be eliminated (this is only needed if the term is to be converted to CNF using the classical method, without constraint generation).

- **`toCNF_Constraints(Term t)`** converts the propositional term `t` (which must be in NNF) into conjunctive normal form, using the approach described in section 3.4.3.

- **`toDIMACS(PrintWriter pw, Term t, Vector mapping)`** converts the propositional term `t` (which must be in CNF) to the DIMACS format and writes the result to `pw`. DIMACS is the input format accepted by SATO to describe propositional formulas (see appendix B). As in the DIMACS format every boolean variable is only represented by an integer number, `toDIMACS` must map the variables contained in the term `t` to numbers. After execution of `toDIMACS`, this mapping can be found in the vector `mapping` — the first element is the variable mapped to 1, etc.

- **`runSATO(Term t, Renaming solution, int maxtime)`** runs SATO to determine a solution for the propositional term `t`. `t` does not need to be in CNF, this conversion is done within `runSATO` by calling the methods described above. If there is a solution, `runSATO` returns `True`, and the solution is included in the `Renaming` object passed to `runSATO` as the second parameter (the `Renaming` object can store a mapping of variables to values). Otherwise, `runSATO` returns `False`. In any case, if SATO does not return a result within `maxtime` milliseconds, it is stopped and an exception is thrown.
  `runSATO()` is a wrapper method that calls the external program SATO (see appendix B). Therefore, the static constants `SATO_STARTUP` and `SATO_OPTIONS` must contain the command to start SATO and the options to be passed to it.

- `PropForm` can report progress information to the caller (e.g. in order to display it in a window in the AUTOFOCUS model browser) via the method `setStatusText` of a "`CheckProgress`" object. For this purpose, the `CheckProgress` object must be made known to `PropForm` using the method **`setCheckProgress`**. The progress information is the same as the trace information, but which messages are actually output is controlled by the static constant `DISPLAY_LEVEL` (rather than `TRACE_LEVEL`).

5.5 The Class `Consts`

Consts	
static Term	AND
static Term	OR
...	...
Term	numeric(int)

Figure 5.6: The Class `Consts`

For convenience, the class `Consts` (see Fig. 5.6) defines `Term` objects for frequently used terms and operators. Using this class, for instance the $\wedge$ operator can be referred to as `Consts.AND`. The method **`numeric(int value)`** returns a `Term` object representing a numeric constant of value `value`.

5.6 The Class `Counterexample`

The solution to a propositional formula as determined by SATO represents an execution sequence corresponding to test data for a certain test case or to a counterexample for a

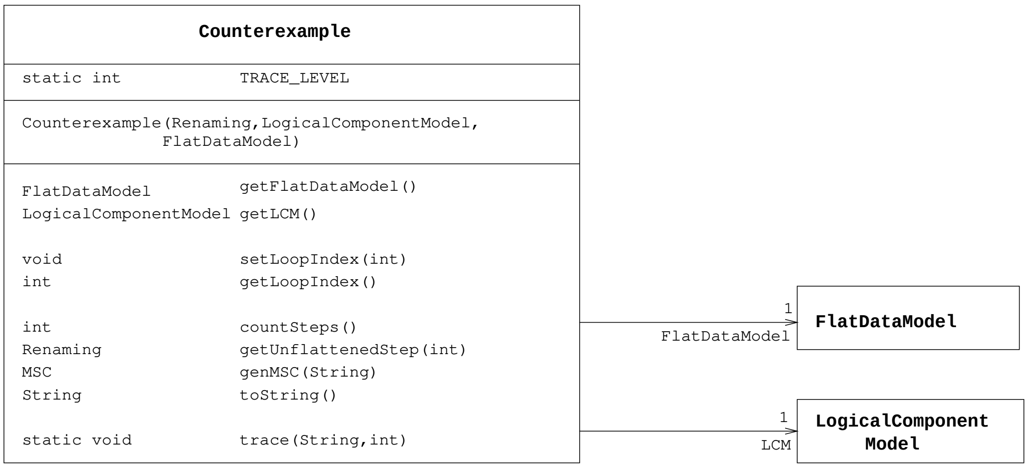


Figure 5.7: The Class `Counterexample`

specification in temporal logic (in case of bounded model checking). However, this solution only consists of the values for the bits of the state variables in the different execution steps. With the help of the class `Counterexample` (see Fig. 5.7), the translation of AUTOFOCUS data types to integers and then to bits is reversed, and the execution sequence is translated to a readable form, like a MSC.

A `Counterexample` object is created by calling the **constructor** with the following arguments:

- the solution calculated by SATO (via `PropForm.runSATO()`) from a propositional formula specifying an execution sequence
- the `LogicalComponentModel` of the component this execution sequence belongs to
- the `FlatDataModel` of the underlying data types

Now, the methods in `Counterexample` can be called to determine information about the execution sequence that corresponds to the solution calculated by SATO:

- **`countSteps()`** returns the number of steps the execution sequence consists of
- **`getUnflattenedStep(int i)`** returns a `Renaming` object that shows the values of all the state variables (i.e., ports, local variables and control states of the automata) at step `i`. The values are already "unflattened", i.e. have the correct AUTOFOCUS data type instead of the integer that data type was mapped to.
- **`genMSC(String name)`** returns a MSC with name `name`, showing the input and output messages occurring in the execution sequence
- **`toString()`** returns a string representation of the execution sequence
- If the execution sequence is a result of the bounded model checking algorithm, it can contain a loop (see section 4.7.2). The bounded model checking algorithm must calculate at which step the loop starts and must set the loop index via the method **`setLoopIndex()`** in `Counterexample`. The MSC generated using `genMSC()` then contains the corresponding loop. The loop index can be retrieved later using **`getLoopIndex()`**.

- **`getFlatDataModel()`** and **`getLCM()`** just return the `FlatDataModel` and `LogicalComponentModel` objects passed to `Counterexample` in the constructor.

5.7 The Class `TSChecker`

```
TSChecker
-----------------------------------------------------------
static int          FLATMODEL_TYPE
static int          TRACE_LEVEL
-----------------------------------------------------------
MSC                 checkTs(Component, RelBag,
                           int,boolean)
MSC                 checkSingleTrans(Component,
                           TransitionSegment, int,
                           int,boolean)
long                getElapsedTime()
String              getMessage()
boolean             done()
void                stop()

void                main(String[])

static void         trace(String,int)
```

Figure 5.8: The Class `TSChecker`

The classes described above provide the general framework to translate AUTOFOCUS specifications to propositional formulas and to solve those formulas, as described in Chapter 3.

Based on this functionality, the class `TSChecker` (see Fig. 5.8) implements the actual specification based test sequence determination for the two test scenarios described in sections 4.2 and 4.3: testing transition sequences and testing single transitions.

There are two ways to use the class `TSChecker` to determine test sequences:

- via an instance of `TSChecker` and its public methods, so the test sequence determination can be used from other classes of the QUEST project or integrated into the AUTOFOCUS model browser
- via the command line, so it can be used on its own as a tool.

Using `TSChecker` from other classes

To use the `TSChecker` from other classes, first an instance of `TSChecker` has to be created with the default constructor. To determine test data for testing transition sequences and single transitions, the `TSChecker` object offers the following methods:

- **`checkTs(Component comp, RelBag transsegs, int maxtime, boolean verbose)`**: tries to find test data corresponding to the execution of the transition sequence `transsegs` in the AUTOFOCUS system specification given by `comp` (as explained in section 4.2) . `transsegs` is a sequence of `Transition-Segment` objects corresponding to the AUTOFOCUS transitions. `checkTs` returns the calculated test sequence as a `MSC` object, or `null` if there is none. If `verbose` is set to `True`, the calculated test sequence is additionally displayed on the standard output.

- **`checkSingleTrans(Component comp, TransitionSegment ts, int k, int maxtime, int verbose)`** Like `checkTs`, but tries to find an execution sequence of maximum length k containing the execution of transition `ts` (see section 4.3).

As the calculations performed by `checkTs` and `checkSingleTrans` might take a long time, the `TSChecker` object contains methods to control their execution and to stop it, if necessary:

- **`getElapsedTime()`** returns the elapsed time since the execution of `checkTS()` resp. `checkSingleTrans()` was started, in milliseconds.
- **`getMessage()`** returns a progress message explaining what task is currently performed
- **`done()`** returns `True` if the execution is finished
- **`stop()`** stops the execution. At the moment, the execution can only be stopped when SATO is running, not during the translation steps performed before.

Using `TSChecker` from the command line

The `TSChecker` object also contains a `main` method, which makes it possible to use it from the command line, as follows:

```
java quest.sato.TSChecker <input-file> <output-file>
                          <qualified-component-name>
```

`<input-file>` must be the name of a QUEST repository file containing the description of a system, `<qualified-component-name>` specifies the component of the system that is to be tested. The user is then offered a numbered list of the transitions of the system and can enter a transition sequence, for which `TSChecker` tries to find corresponding test data. If test data is found, it is displayed on the console and added to the system description as a MSC. The new system description containing the MSC that shows the test data is then output to the QUEST repository file `<output-file>`.

In the current implementation, test data determination for single transitions can not yet be invoked via the command line.

5.8 The Class `BMChecker`

The class `BMChecker` is structured in the same way as the class `TSChecker`. It implements the test data determination for test scenarios described by properties in temporal logic, respectively the bounded model checking algorithm (see section 4.7).

Using `BMChecker` from other classes

`BMChecker` can be used from other classes like `TSChecker`, by creating a `BMChecker` object. However, as properties in temporal logic can be part of an AUTOFOCUS/ QUEST system description, they do not have to be specified explicitly when performing the test data determination:

- **`checkComponent(Component comp, int k, long maxtime, boolean verbose)`**: for every property p associated to `comp`, checkComponent tries to find an execution sequence showing that there is an execution of the system with maximum length `k` that does *not* fulfill this property. If there is such an execution sequence, the corresponding MSC is added to the component `comp`, as a counterexample for the property p. The parameters `maxtime` and `verbose` have the same meaning as in the methods of `TSChecker` (see section 5.7).

In addition, `BMChecker` offers the same methods to control the execution of `checkComponent()` as `TSChecker`: `getElapsedTime()`, `getMessage()`, `done()` and `stop()`. Moreover, `BMChecker` can automatically report its progress to the caller via a `CheckProgress` object, which must first be made known to `BMChecker` using the method `setCheckProgress()` (as in `PropForm`, see section 5.4).

Bounded model checking using the class `BMChecker` has already fully been integrated into the AutoFocus model browser and can be invoked via its graphical user interface (see the QUEST documentation [BLS99] for details).

Using `BMChecker` from the command line

`BMChecker` can also be called from the command line:

```
java quest.sato.BMChecker <input-file> <output-file>
                          <qualified-component-name>
                          <bound>
```

This command tells `BMChecker` to read an AutoFocus system description from the file `<input-file>` and perform `checkComponent()` on the system component given by `<qualified-component-name>`. `<bound>` specifies the maximum length for the determined counterexamples. The new system description containing the counterexamples is output to `<output-file>`.

5.9 The Class `SMVExport`

Finally, the class `SMVExport` makes it possible to translate an AutoFocus system to the SMV input language, as described in section 3.2.6. Again, like `BMChecker` and `TSChecker`, `SMVExport` can be used from other classes or from the command line.

For the use from other classes, `SMVExport` offers the method **`export(Component comp, String export_filename)`**. This method translates the system specification of the AutoFocus component `comp` to the SMV input language and writes the result to a file with the name `export_filename`. Also, all properties associated to the component `comp` are translated to SMV specifications (section `SPEC`) and appended to that file. As bounded model checking, export to SMV via the class `SMVExport` has been fully integrated into the AutoFocus model browser and is available via a menu item. The browser also contains functionality to read back the counterexample generated by SMV and translate it to a MSC.

From the command line, `SMVExport` is used as follows:

```
java quest.sato.SMVExport <input-file>
                          <qualified-component-name>
                          <output-file>
```

With this command, a specific component (given by `<qualified-component-name>`) of the system specification contained in the QUEST repository file `<input-file>` is translated to the SMV input language and the result is written to `<output-file>`.

Chapter 6

Experimental Results

In this chapter, we present some performance statistics obtained with the Java implementation described in Chapter 5 and thus evaluate the applicability of specification based test sequence determination with SATO for the different test scenarios. In addition, we include a comparison of bounded model checking (see section 4.7) and conventional model checking with SMV.

As an example system we chose the `Till` component of the cash point system (see Fig. 2.3). All performance figures were obtained on a SUN UltraSparc 2 with 1 GB of main memory and two 200MHz processors.

6.1 Determining Input Data for Transition Sequences

The first application of specification based test sequence determination we described in Chapter 4 was determining input data for transition sequences. As benchmark we took the first $5, 10, \ldots, 100$ transitions of the transition tour we described in section 4.2.3 for the `Till` component (this transition tour only consists of 34 transitions, but as it is a transition tour, we can start from the beginning again after the 34th transition).

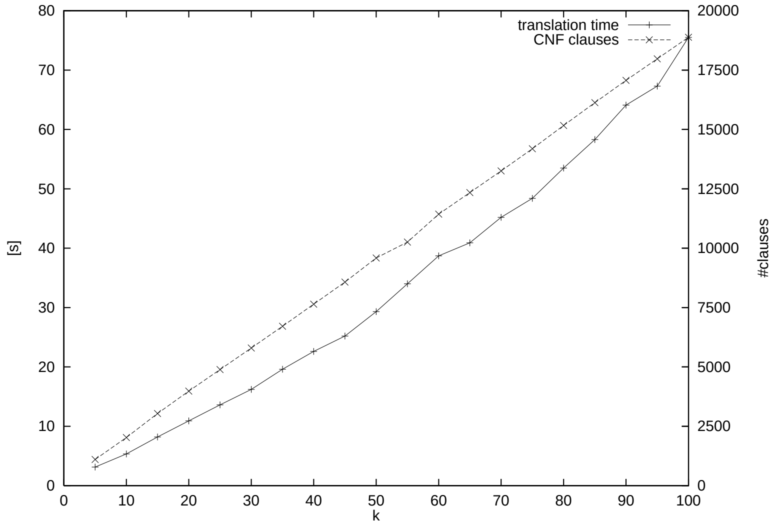


Figure 6.1: Translation of a Transition Sequence to CNF

Fig. 6.1 shows the execution time (average of 5 test runs) to convert the transition sequence into the propositional formula $M \wedge P$, and the number of clauses of the resulting formula after conversion to conjunctive normal form. The number of clauses directly influences the time needed by SATO to then determine input data for the transition sequence by calculating a solution for $M \wedge P$.

As to be expected, both the execution time for the translation and the number of clauses grow linearly with the length of the transition sequence. This is because of the structure of the formula $M \wedge P$, which consists of a conjunction of one subterm for each transition in the sequence. The overall execution time of the translation is in an acceptable range (less than 80 seconds for a transition sequence of 100 transitions).

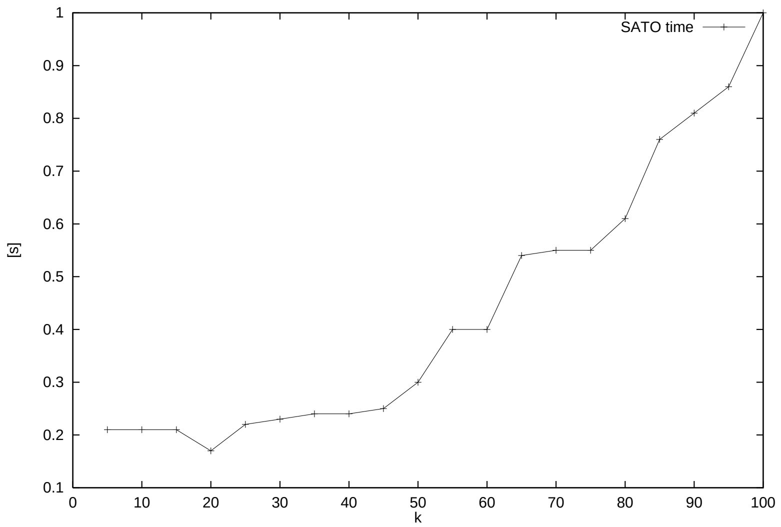


Figure 6.2: Calculating Input Data with SATO

Fig. 6.2 shows the execution time SATO needed to determine input data for the transition sequences, after they were translated to conjunctive normal form. In this figure, we can observe the exponential performance behaviour of SATO, depending on the length of the transition sequence (and thus the size of the formula to be solved). A remarkable fact is that in spite of the exponential complexity, SATO still takes not more than one second even in case of a 100-step transition sequence (for practical applications, the transition sequences to be considered will usually be much shorter). In addition, the SATO execution time can almost be neglected in comparison to the time needed to translate the execution sequence to a propositional formula beforehand. This is because SATO is implemented in C, whereas our translation is implemented in the much slower programming language Java.

Thus, our approach is very suitable to determine input data for transition sequences. This is especially the case if (as in our example) the system to be tested only consists of one automaton — which is the main application area for input data determination for transition sequences. The optimisation shown in section 4.2 leads to compact formulas that can easily be solved by SATO, even for transition sequences of length up to 100.

6.2 Testing Single Transitions

As a second benchmark, we used SATO to determine execution sequences of length k containing the execution of a certain transition at some step, as described in section 4.3. We chose the transition (14) of the `Till` component (see Fig. 2.3), corresponding to the successful termination of a money withdrawal transaction.

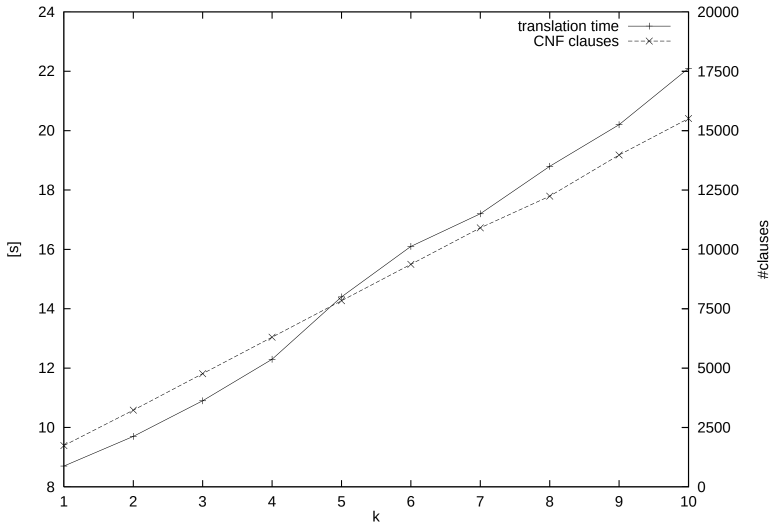


Figure 6.3: Translation to CNF for Single Transition Test

Fig. 6.3 again shows the translation time to $M \wedge P$ for execution sequences of different lengths k and the number of its clauses. As the optimisation of $M \wedge P$ for determining input data for transition sequences given in section 4.2 cannot be applied anymore, the size of $M \wedge P$ grows much faster. For an execution sequence of length $k = 10$, it already consists of approximately 15000 clauses.

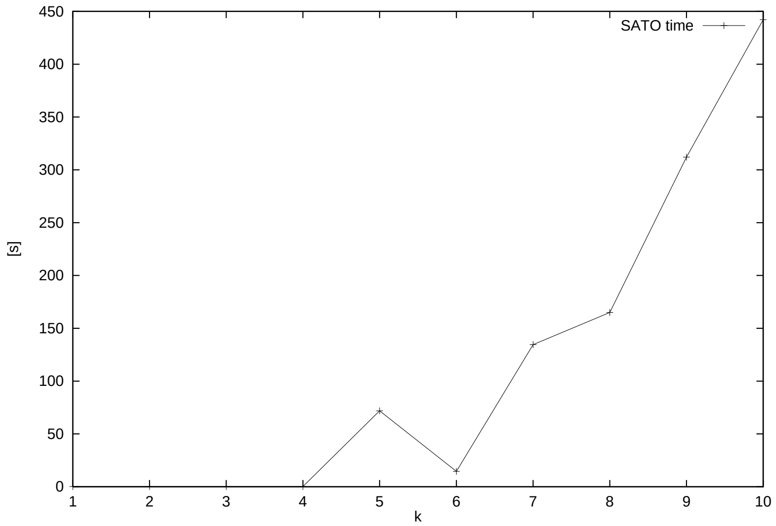


Figure 6.4: Determining Single Transition Test Data with SATO

The SATO execution time to determine a solution to $M \wedge P$ (and thus to find an execution sequence containing the specified transition) is depicted in Fig. 6.4. In this benchmark, SATO was terminated if its execution took longer than 600 seconds, and an average of 10 test runs was calculated. From Fig. 6.4, one can see that the formula $M \wedge P$ is not only much larger than in case of determining input data for transition sequences, but also much more complex for SATO to solve. Though the number of clauses for $k = 10$ is actually less than when determining input data for a transition sequence of length 100, the average execution time SATO takes to solve it is greater by orders of magnitude.

An important fact to mention is that for higher values of k, the SATO execution time varies considerably from one test run to another — for instance, for $k = 5$ we measured times ranging from $0.2s$ to $> 600s$ (time limit exceeded). The reason for this is that SATO works by "guessing" values for the variables in $M \wedge P$ to determine a solution. If it makes good guesses at the beginning, a solution can be found very fast, and vice versa. This explains the odd performance behaviour between $k = 5$ and $k = 6$ (the execution time decreases): we only took 10 measurements and SATO just had more "lucky guesses" for $k = 6$.

We can conclude that our approach is very suitable for testing single transitions, but only if the maximum length k of the execution sequence we consider is not too large. In our example, for $k \leq 4$, the execution times we measured were less than half a second, but then increased rapidly. In addition, if SATO does not return a result after some time, it is often a good idea to stop it and run it again. Because of the great variation of the execution times it might calculate a solution much faster in the next run.

6.3 Bounded Model Checking and Comparison to SMV

Finally, we evaluated the performance of bounded model checking of AUTOFOCUS systems, i.e. determining execution sequences of maximum length k that fulfill a certain specification given in temporal logic (see section 4.7). As an example, we again took the `Till` component. We used the specification $\mathbf{A}\Diamond\mathbf{False}$, stating that there is no infinitely long execution sequence. Of course this is false, so as a result, we obtain an infinitely long execution sequence of the system (i.e., an execution sequence containing a loop).

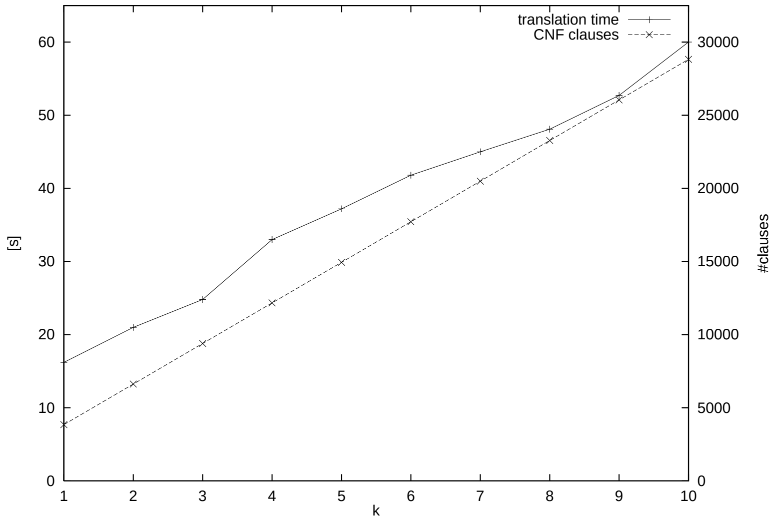


Figure 6.5: Translation to CNF for Bounded Model Checking

Fig. 6.5 and 6.6 show a similar performance behaviour as observed for the single transition test. The formula $M \wedge P$ is about twice as large, and the SATO execution time is even more unpredictable, showing great variations between the different runs.

The `Till` component was also translated to the SMV input language (as described in 3.2.6), and we determined a counterexample for $\mathbf{A}\Diamond\mathbf{False}$ using conventional BDD-based model checking with SMV. The line in Fig. 6.4 shows the corresponding SMV execution time.

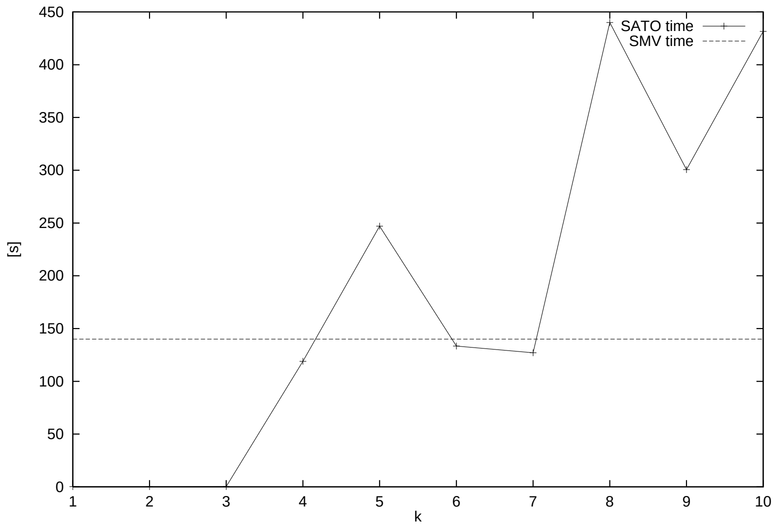


Figure 6.6: Bounded Model Checking with SATO

Whereas SATO is considerably faster when only short counterexamples are considered (i.e., k is small, in this case $k \leq 3$), it quickly loses its advantage with larger values of k. In addition, the SMV execution time is much more predictable than the execution time of SATO. Moreover, a strong argument in favour of SMV is that we do not have to fix a bound k — if the specification is false, SMV always returns a counterexample of the necessary length.

Thus we can resume that in our example, bounded model checking with SATO is very useful to quickly obtain counterexamples of a small, limited maximum length. If we cannot fix a bound or the expected counterexamples are large, SMV is preferable.

6.4 Conclusion

The performance statistics we showed in this chapter prove the practical applicability of our approach to determine test sequences by solving propositional formulas. In particular, input data for transition sequences can be calculated very efficiently using this method, even for transition sequences of lengths up to 100. But also determining test data to test single transitions or bounded model checking turned out to be possible very fast, provided k is not too large. For small values of k, our approach by far outperforms conventional model checking with SMV.

Of course, our performance evaluation is not exhaustive. To obtain reliable information about the performance behaviour of specification based test sequence determination via SATO, it is necessary to examine other AUTOFOCUS specifications (in particular, specifications consisting of multiple components), to compare input data determination for different transition sequences, to compare bounded model checking for different LTL predicates etc. However, preliminary statistics obtained for other specifications up to now always showed a similar behaviour as encountered for the `Till` component — very fast execution times for small values of k, which from a certain value of k increased greatly and varied considerably between the different runs of SATO.

Chapter 7

Conclusion

In this thesis, we showed how test data for embedded system tests can be determined from a formal specification. The logical model of AUTOFOCUS specifications we developed for this purpose is generally applicable and can also be of use for many other problems in the area of specification, verification and testing — for example, it leads to an efficient translation to the SMV input language and is the basis for a bounded model checker for AUTOFOCUS.

In addition, we showed different applications of specification based test data determination to aid testing of embedded systems, for instance determining input data for executing transition sequences or particular single transitions in the specification. This way, it was possible to systematically test an implementation of an embedded system for its correctness with respect to the specification: the tester selects for example a certain critical transition in the specification, or a transition sequence covering a number of transitions, then determines the corresponding input data and expected output data from the specification and finally checks if the behaviour of the implementation is as expected when the calculated input data is passed to it.

We also explained our implementation of a framework for specification based test sequence determination in the programming language Java, which is now part of the AUTOFOCUS/QUEST toolset and also provides a SMV translation and a bounded model checker. The performance evaluation we carried out demonstrated that our methods are well applicable in practice. In particular, determining input data is possible for transition sequences consisting of more than 100 transitions, and the bounded model checking algorithm performs considerably faster than conventional model checking for small enough bounds on the lengths of the counterexamples.

7.1 Limits

However, our approach to determine test data is restricted to a limited class of system specifications: all used data types have to be finite and there must not be any recursive function definitions. Otherwise, it would not be possible to represent the state space of the system by a fixed number of boolean variables and thus to determine test data as a solution to a propositional formula containing these variables. In addition, we assumed that the system specification is deterministic. Non-deterministic specifications allow more than one possible behaviour of the implementation in the same situation (i.e., when the same test data is passed to it) and make testing very difficult. We presented some suggestions how to handle non-deterministic specifications, but there are still many open questions in this area.

Another important point is that the length k of the execution sequences to be considered should not be too large. The size of the propositional formula that has to be solved is proportional to k, and because of the exponential performance behaviour of satisfiability solvers like SATO, obtaining a solution is impossible if k exceeds a certain range. The maximum possible value for k is dependent on the complexity of the system specification. For the `Till` component of our bank example, a system of medium complexity, SATO execution time started to rise drastically from about $k \geq 6$, but it was still possible to calculate execution sequences for up to $k = 10$ or more. The input data determination for transition sequences however is much more efficient due to our optimisation. As mentioned above, it even works very fast for transition sequences consisting of more than 100 steps.

7.2 Further Work

Our approach is only a first step in supporting system testing with the help of a logical system model. There are many possibilities for improvements and areas for further work:

- Apart from the six applications for test data determination we showed in chapter 4 — e.g. test data determination for transition sequences or for testing single transitions — there should be many other test scenarios where our approach can be put to use. In particular, test data determination for EETs might be an interesting topic, which we examined only very shortly and did not include in our implementation.
- As mentioned in section 4.3.2, one could also use test sequence determination via SATO to optimize system specifications, for example to check if transitions in the specification are really needed or can be omitted safely because they are not executable anyway.
- The performance evaluation we gave in chapter 5 is also only preliminary. Many more different systems and test cases have to be considered to obtain reliable performance statistics and to see for which kinds of systems and up to which bounds k for length of the execution sequences our approach is feasible.
- SATO is not the only tool that can be used to solve propositional formulas. Other tools, like PROVER [Bor97] or SEMPROP [Let99] could be connected to AUTOFOCUS, and the performance behaviour could be compared with SATO.
- Even when using SATO, there are various possibilities to improve the execution time. In chapter 6, we explained that the execution time of SATO varies considerably, so one could for example run multiple instances of SATO in parallel and wait for the first instance to return a solution. Also, SATO itself could be parallelized.
- Lastly, it would be challenging to examine how to overcome the limitations of the system specifications that can be used for testing. On the one hand, different strategies to test non-deterministic systems could be evaluated. On the other hand, infinite systems can be treated by translating them to finite systems using abstraction techniques [Mül98]. One could also think of leaving the AUTOFOCUS data types and functions in the logical model of the execution sequence instead of translating it to a propositional formula, and trying to calculate execution sequences directly using a theorem prover instead of the satisfiability solver SATO.

Bibliography

[Bar91] Thomas C. Bartee. *Computer Architecture and Logic Design*. McGraw-Hill, Inc., 1991.

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *TACAS99*, 1999.

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proc. of the Fifth Annual Symposium on Logic in Computer Science*, 1990.

[BDD+92] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The Design of Distributed Systems – an Introduction to FOCUS. Technical Report TUM-I9203, Munich University of Technology, Institute of Computing Science, January 1992.

[BJR97] Grady Booch, Ivar Jacobson, and Jim Rumbaugh. *UML Summary*. Rational Software Cooperation, 1997.

[BLS99] Peter Braun, Heiko Lötzbeyer, and Oscar Slotosch. Quest Users Guide, 1999. Internal Paper, Munich University of Technology, Institute of Computing Science (Department Prof. Broy).

[Bor97] A. Boralv. The industrial success of verification tools based on stalmarck's method. In *International Conference on Computer-Aided Verification (CAV'97)*, number 1254 in LNCS. Springer Verlag, 1997.

[CDA98] J. C. Corbett, M. B. Dwyer, and G. S. Avrunin. Property specification patterns for finite-state verification. In *2nd Workshop on Formal Methods in Software Practice*, 1998.

[DP60] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the Association of Computing Machinery*, (7):201–215, 1960.

[Fit90] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, 1990.

[FMR+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+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.

[HP96] G. Holzmann and D. Peled. The State of SPIN. In *Proc. of CAV'96*, pages 385–389, New Brunswick, NJ, 1996. Springer Lecture Notes in Computer Science, #1102.

[HS97] Franz Huber and Bernhard 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] Franz Huber, Bernhard Schätz, and Geralf Einert. Consistent graphical specification of distributed systems. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME '97: 4th International Symposium of Formal Methods Europe, Lecture Notes in Computer Science 1313, pp. 122 - 141*. Springer, 1997.

[IT93] ITU-T. *Recommendation Z.100, Specification and Description Language (SDL)*. ITU, 1993.

[ITU96] ITU. ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU-TS, Geneva, 1996.

[Jon93] M. P. Jones. *An Introduction to Gofer*. August 1993.

[KMSS98] T. Klein, O. Müller, S. Sadeghipour, and O. Slotosch. Derivation of Adequate Test Methods for the QUEST Process Model. Technical report, Daimler Benz AG / Technische Universität München, 1998.

[Let99] Reinhold Letz. SEMPROP, 1999. Propositional SAT solver developed at Munich University of Technology. For further information, contact R. Letz (mailto:reinhold.letz@in.tum.de).

[McM92] K. L. McMillan. *The SMV system (DRAFT)*. Carnegie-Mellon University, February 1992. Available together with the SMV tool via: http://almond.srv.cs.cmu.edu/afs/cs.cmu.edu/project/modck/pub/www/smv.html.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.

[Mil80] Robin Milner. *A Calculus of Communcating Systems*. Springer, Berlin, 1980.

[Mül98] Olaf Müller. *A Verification Environment for I/O-Automata Based on Formalized Meta-Theory*. PhD thesis, Institut für Informatik, Techn. Univ. München, 1998.

[PG86] D. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, (2):293–304, 1986.

[PS99] J. Philipps and O. Slotosch. The Quest for Correct Systems: Model Checking of Diagr amms 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.

[SK98] S. Sadeghipour and T. Klein. Testmethoden für AUTOFOCUS Spezifikationen. Technical report, Daimler Benz AG, Forschungsabteilung FT3/SM, 1998.

[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.

[Slo99] O. Slotosch. The FM99 Banking System with Quest, 1999. Available via `http://www4.in.tum.de/proj/quest/`.

[Stå89] G. Stålmarck. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula. Swedisch patent no. 467 076 (1992), US patent no. 52767897 (1994), European patent no. 0404454 (1995), 1989.

[Ura92] H. Ural. Formal methods for test sequence generation. *Computer Communications*, 15(5):311–325, 1992.

[Zha97] H. Zhang. SATO: An efficient propositional prover. In *International Conference on Automated Deduction (CADE '97)*, number 1249 in LNAI, pages 272–275. Springer Verlag, 1997. See also `http://www.cs.uiowa.edu/~hzhang/sato.html` (SATO homepage and ftp download).

Appendix A

AutoFocus specification of the Cash Point System

In this appendix, we include the complete AutoFocus specification of the cash point example system explained in chapter 2.

A.1 System Structure Diagrams

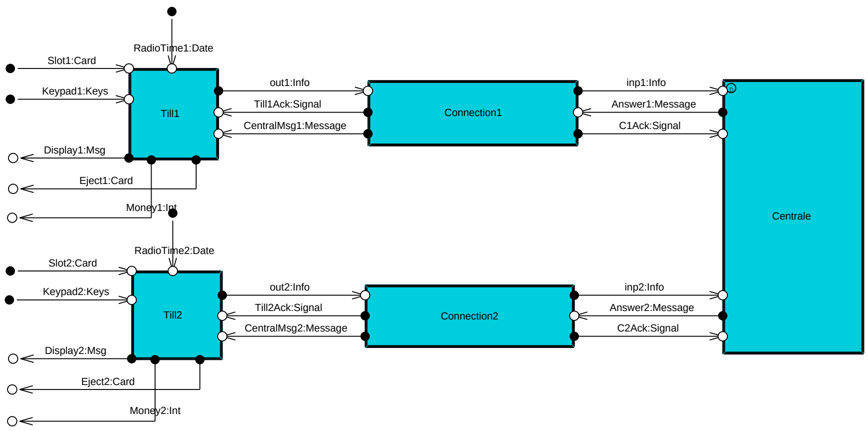


Figure A.1: SSD for Banking System

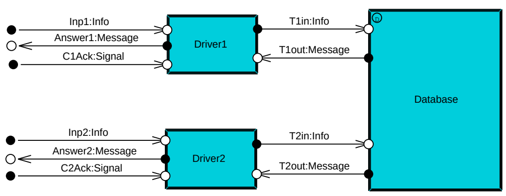


Figure A.2: SSD for `Centrale`

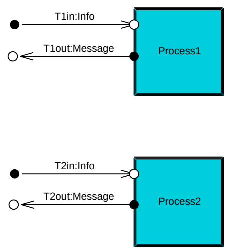


Figure A.3: SSD for `Database`

A.2 State Transition Diagrams

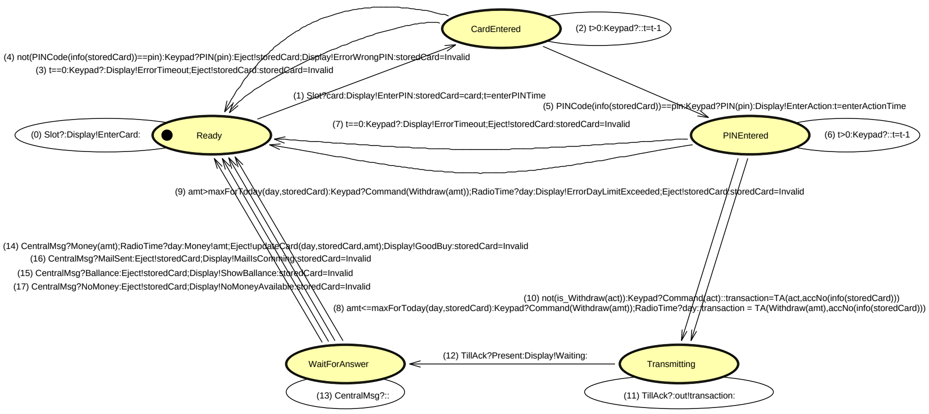


Figure A.4: STD for `Till`

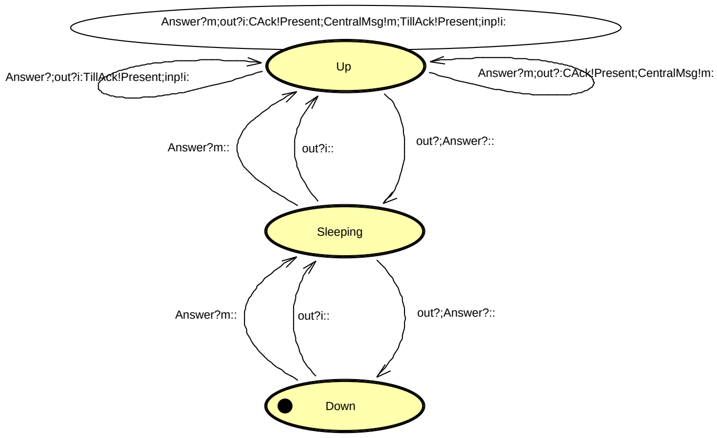


Figure A.5: STD for `Connection`

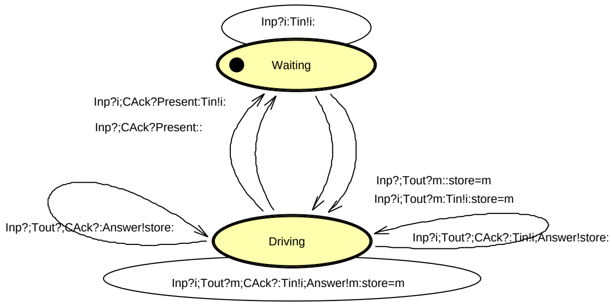


Figure A.6: STD for `Driver`

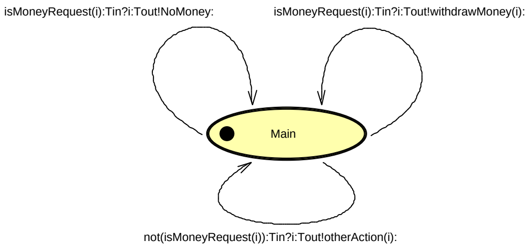


Figure A.7: STD for `Process`

A.3 Data Type Definitions

```
module CARD =
import DATE;
data Cardinfo = CI(PINCode:Int,accNo:Int);
data Card = Invalid | Valid(info:Cardinfo,dayAmount:Int,Date);
const aCard = Valid(CI(0,0),0,mkDate(0));
const maxAmount=2;
fun maxForToday(day,Valid(ci,amt,lastDay)) =
      if (lastDay == day)
      then maxAmount-amt
      else maxAmount
      fi;
fun updateCard(day,Valid(ci,amt,lastDay),withdraw) =
      if (lastDay == day)
      then Valid(ci,amt+withdraw,day)
      else Valid(ci,withdraw,day)
fi;


end
module DATE =
data Date = mkDate(Int);
const enterPINTime=2;
```

```
const enterActionTime=1;


end

module MESSAGE =
import TRANSACTION;
data Message = Money(Int)  | NoMoney | Ballance | MailSent;
data Msg = EnterCard | EnterPIN | EnterAction
                | Waiting | NoMoneyAvailable | MailIsComming | Show-
Ballance | GoodBuy
                | ErrorTimeout | ErrorWrongPIN | ErrorDayLimitExceeded;
fun withdrawMoney(TA(Withdraw(x),acc))=Money(x);
fun otherAction(TA(SendMail,acc))=MailSent
 |  otherAction(TA(ViewBallance,acc))= Ballance;


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


end

module SIGNAL =
data Signal =Present;


end
```

Appendix B

SATO

This appendix describes the usage of the propositional satisfiability solver SATO, and one of its input languages, the DIMACS format. SATO can be downloaded for free from the internet (see [Zha97]).

B.1 The DIMACS format

DIMACS is one of the formats describing propositional formulas in CNF that can be read by SATO (and also by other SAT solvers like PROVER). A DIMACS file has a very simple structure: it starts with a header line

```
p cnf <#vars> <#clauses>
```

which gives the number of variables and clauses in the formula. The variables are then represented by consecutive integer numbers starting from 1. The rest of the file contains the clauses of the formula, one after the other. Each clause is represented by a sequence of integer numbers corresponding to the literals in the clause: a positive integer number stands for a variable, a negative integer number for its negation. All the clauses are terminated with "0".

In addition, a DIMACS file can contain lines starting with "`c`", which are treated as comments and ignored. For example, the comments can be used to show which variable of the original formula is represented by which integer number in the DIMACS file.

Example

As an example, the formula
$(\mathbf{not}(B) \vee C \vee \mathbf{not}(D)) \wedge A \vee \mathbf{not}(C)) \wedge (A \vee C \vee D) \wedge (A \vee B) \wedge (\mathbf{not}(A) \vee \mathbf{not}(B) \vee \mathbf{not}(C))$
can be translated into the following DIMACS file:

```
c 1 = A
c 2 = B
c 3 = C
c 4 = D

p cnf 4 5
-2 3 -4 0
1 -3 0
1 3 4 0
1 2 0
-1 -2 -3 0
```

B.2 Usage of SATO

The usage of SATO is very easy: just type

```
sato <dimacs_file_name>
```

and SATO reads the formula described in the corresponding DIMACS file and tries to calculate a valuation of the variables that satisfies the formula. The valuation is given as the indizes of the variables that should be set to `True`.

Example

For the example formula shown above, sato gives the following output (corresponding to the solution $A = \textbf{True}, B = \textbf{False}, C = \textbf{False}, D = \textbf{False}$):

```
bash-2.01$ sato example.dimacs
------- SATO 3.2 on Edwin ------
The job "sato example.dimacs" started at Sun Jan 23 22:50:24 2000

Input file "example.dimacs" is open.
Reading clauses in DIMACS's format.
Max_atom = 4, Max_clause = 5
There are 5 input clauses.
The maximal index of propositional variables is 4.
The mallocated memory is      64.00 Kbytes.
There are 5 clauses in the trie (2 are non-Horn).

Model #1: (indices of true atoms)

    1

The number of found models is 1.

There are 5 branches (1 succeeded, 1 failed).

The search tree path is:
 ( s1 f-2 f-3 f-4 )

---------------- Stats ----------------
run time (seconds)                 0.00
  build time                       0.00
  search time                      0.00
mallocated (K bytes)              86.10
---------------------------------------
The job "sato example.dimacs" ended at Sun Jan 23 22:50:24 2000
```