Specification Based Test Sequence Generation with Propositional Logic

G. Wimmel, H. Lötzbeyer, A. Pretschner, and O. Slotosch
Institut für Informatik, Technische Universität München
Arcisstraße 21, 80290 München, Germany
`www4.in.tum.de/~{wimmel,loetzbey,pretschn,slotosch}`

Abstract

In the domain of concurrent reactive systems, much work has been devoted to (semi-) automatically validating a system's correctness. A novel approach to the automated generation of test sequences is presented. It may be used for both glass box testing a specification and black box testing an implementation (SW/HW). Finite system models specified within the CASE tool AutoFocus as well as user-friendly test case specifications are automatically translated into propositional logic and fed into the propositional solver SATO. Results are interpreted as I/O traces (test sequences) of the system, and may be displayed as message sequence charts. A small example illustrates the basic ideas as well as the method's advantages and shortcomings. The testing process is integrated into an overall development process. Main contributions include the implementation of a tool for graphical specification of test cases and the description of an efficient method to fully automatically compute test sequences as well as its integration into the same CASE tool.

Keywords. Automatic Test Case Generation, Reactive Systems, Propositional Logic, CASE, Message Sequence Charts, Validation

1 Introduction

A typical development process for reliable concurrent reactive systems consists of possibly cyclic iterations of requirements analysis, system design, implementation, and validation. Such a development process should support validated transitions from requirement specifications to system models and from abstract design specifications to detailed implementations. There are two common methods of validation, namely verification and testing (simulation). Verification, understood as establishing logical implication, ensures – with mathematical rigor – a model's correctness, whereas testing usually is restricted to exemplary system runs. In the face of complexity as well as acceptance problems in industry as encountered by verification techniques such as model checking and theorem proving [25], the increasing industrial need for testing becomes understandable.

Testing can be applied to both system models (glass box) and actual implementations (hardware with generated code, black box). A *test case specification* is the formalization of some test purpose (e.g., possibly partial I/O traces where certain input as well as output values can be specified, reachability of a set of states, firing conditions for sets of transitions, etc.; see [18] for a precise terminological framework). A *test sequence* is an I/O trace that satisfies a given test case specification. Specification of test cases is done during design or, possibly interactively (rapid prototyping), during the validation phase. Testing a system (i.e., simulating it under certain user-defined constraints) also helps the system designer to better understand and debug it.

Overview. This paper presents an approach to generating test sequences on the grounds of the system's semantics in terms of propositional logic. Its remainder is organized as follows. Section 2 describes the CASE tool AutoFocus that has been used for the specification of different views on system models as well as test cases. Inspired by the work of Biere et al. on bounded model checking [4], section 3 describes the automated translation of AutoFocus specifications into propositional logic. The solver SATO [31], based on the Davis-Putnam approach to satisfiability in propositional logic, is then used for determining test sequences. These sequences can be displayed graphically as a time-synchronous variant of Message Sequence Charts (MSCs [14]). This approach to determining test sequences can be seen as a special application of bounded model checking. However, a set of common test case specifications can be formulated without referring to the Linear Time Logic (LTL) which results in efficiency gains, and more importantly, in a user-friendly graphical description of properties that can be used to specify test cases. On the other hand, there are test case specifications that cannot be translated into pure LTL without altering the model (and thus increasing its complexity; e.g., talking explicitly about certain data driven transitions). The specification of an Automated Teller Machine (ATM) illustrates the translation as well as the application of several model-based test case specifications (section 4). The paper concludes with a description of related work and a discussion of this new approach.

Contributions. The main contributions of this work may be summarized as follows. First, an automated translation of concurrent systems as well as certain classes of test cases into propositional logics is presented. This translation is used for the computation of test sequences by means of an efficient propositional solver (SATO). Furthermore, it gives rise to automatically performed optimizations such as model slicing. As expected, the use of a specialized solver for a subclass of all testing problems yields more efficient results than traditional general-purpose approaches (e.g., general (bounded) model checking). Second, a tool-supported intuitive graphical specification of test cases is discussed and used in connection with the above translation. It is shown how Message Sequence Charts can be used for various classes of test cases and argued why, for these classes, they might be a superior language for the specification of such test cases.

2 Modeling with AutoFocus

AutoFocus [12, 13] is a tool for graphically specifying embedded systems on the grounds of a simple, formally defined semantics. It supports different views on the system model: structure, behavior, interaction, and data type view. Each view concentrates on a fixed part of the specified model.

Structural view: SSDs. In AutoFocus, a distributed system is a network of components, possibly connected one to another, and communicating via so-called channels. The partners of all interactions are components which are specified in *System Structure Diagrams* (SSDs). Figure 1 shows a typical SSD. In this static view of the system and its environment, rectangles represent components and directed lines visualize channels between them. Both of them are labeled with a name. Channels are typed and directed, and they are connected to components at special entry and exit points, so called *ports*. Ports are visualized by filled and empty circles drawn on the outline (the *interface*) of a component. As SSDs can be hierarchically refined, ports may be connected to the inside of a component. Accordingly, ports which are not related to a component are meant to be part of unspecified components which define the *outside world* and thus the component's interface to its environment.

Behavioral view: STDs. The *behavior* of an AutoFocus component is described by a *State Transition Diagram* (STD). Figure 2 shows typical STDs. Initial states are marked with a black dot. An STD consists of a set of *control states*, *transitions* and *local variables*. The set of local variables builds the automaton's *data state*. Hence, the internal state of a component consists of the automaton's control as well as its data state.

A transition can be complemented with several annotations: a label, a precondition, input statements, output statements and a postcondition, separated by colons. The precondition is a boolean expression that can refer to local variables and transition variables. Transition variables are bound by input statements, and their life-cycle is restricted to one execution of the transition. Input statements consist of a channel name followed by a question mark and a pattern. An output statement is a channel name and an expression separated by an exclamation mark. The expression on the output statement can refer to both local and transition variables.

A transition can *fire* if the precondition holds and the patterns on the input statements match the values read from the input. After execution of the transition the values in the output statements are copied to the appropriate ports and the local variables are set according to the postcondition. Actually the postcondition consists of a set of actions that assign new values to local variables, i.e., the assignments set the automaton's new data state.

Communication semantics. AutoFocus components have a common global clock, i.e., they all perform their computations simultaneously. The cycle of a composed system consists of two steps: First each component reads the values on its input ports and computes new values for local variables and output ports. After the clock tick, the new values are copied to the output ports where they can be accessed immediately via the input ports of connected components and the cycle is repeated. This results in a *time-synchronous* communication scheme with buffer size 1.

Interaction view: MSCs. Message Sequence Charts (MSCs) are used to describe the interaction of components. In contrast to Message Sequence Charts as defined in [14], AutoFocus MSCs refer to time-synchronous systems. In the following, the term MSC always denotes these time-synchronous sequence charts. Progress of time is explicitly modeled by ticks which are represented by dashed lines. All actions between two successive ticks are considered to occur simultaneously, i.e., the order of these actions is meaningless. An action in an MSC describes a message that is sent via a channel from one component to another. This is denoted by a horizontal arrow from the source to the destination component. Internal messages between two components are annotated with the channel and the contents of the message separated by a dot. Annotations on external messages do not refer to channels but rather to external ports of the component. To illustrate this fact, the port name and the message value are delimited by ! (send) or ? (receive) in analogy to transition annotations in STDs.

Note that it does not take any time to transfer messages. Time is consumed during the ticks, when the computations of all components are performed synchronously. As a consequence, the output values cannot depend on the input values of the same time slice and each component always needs a tick for the computation of new output values. Figure 4 shows typical exemplary system runs.

Datatype view: DTDs. For the specification of user defined data types and functions AutoFocus provides DTDs. The definitions in DTDs are written in a Gofer-like functional style. Table 1 contains an example.

Example. A simple ATM system (Fig. 1) will serve as an example for modeling, translation, and testing. The system consists of three components: a timer, a central data base, and a till component (the actual ATM). Channel user serves

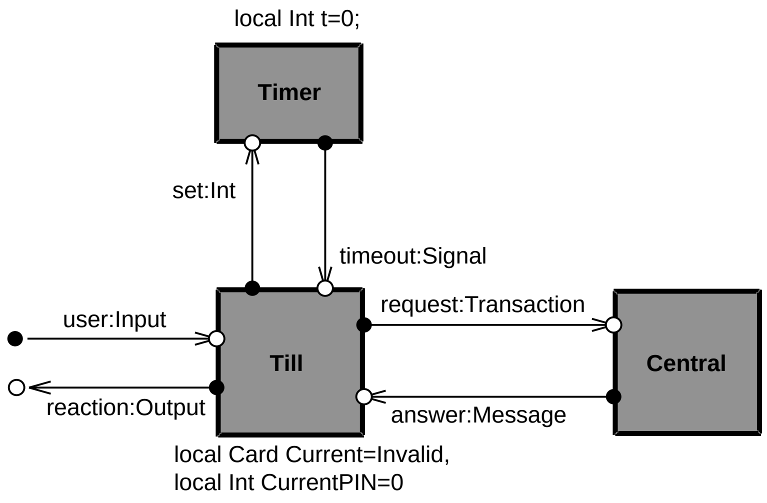


Figure 1: System structure

as the ATM's input interface; a Card may be entered into the Slot, a function key FunKey with some associated Action may be pressed, or a PIN (an Integer) may be entered. Table 1 shows the associated data types. Users get the system's reaction via channel reaction (request for action, issuing money, displaying the balance). The timer component ensures that after a certain time the card is returned (e.g., in case something went wrong). Finally, the central database gets a request from the Till component on channel request and reacts accordingly (e.g., transmit balance, issue money, etc., see Table 1 for details). The system's behavior is depicted in Fig. 2.

Signal =	Present
Card =	Invalid | Valid(getPIN:Int, Acc:Int)
Action =	Withdraw(Int) | ViewBalance
Input =	Slot(Card) | FunKey(Action) | PIN(Int)
Transaction =	TA(Action, acc:Int, pin:Int)
Message =	NoMoney | Money(amt:Int) | Balance
Output =	enterPIN | enterCard | enterAction | timeoutError(Card)
	| byebye(money:Int, Card) | ViewBal(Card) | errorWrongPIN

Table 1: Data types in the ATM model

Note that a pattern matching expression inputChannel? without a pattern after the question mark requires inputChannel not to carry any value. This is different from inputChannel?X which allows inputChannel to carry an arbitrary value. Also note that in this simple example, no accounts are maintained, and whenever a valid card is entered, any amount of money may be withdrawn. Furthermore, a reaction ViewBal(Card) indicates that *some* balance is displayed and that the card is returned. For simplicity's sake, the specification allows a user to enter new cards before a card is returned. This could be avoided by adding an extra variable that stores if a card has been entered (and rejecting subsequent ones). If in this model channel user carries a value of type Card even though there already is a card in the system, the second card might be thought as being immediately rejected.

3 Encoding into Propositional Logic

Once each component of an AutoFocus system model has been assigned an STD, the behavior of the system — as given by its possible I/O traces — is completely specified. A test sequence is an executable I/O trace satisfying a certain test case specification. The presented approach to determining test sequences consists of translating a system specification into a propositional formula describing all its possible executions of a certain length, given as a sequence of global system states. Likewise, test case specifications are translated into a predicate over execution sequences. A solution to the conjunction of these two propositional formulas is then computed using standard satisfiability solvers such as SATO [31] and results in an execution sequence satisfying the test case specification. From this execution sequence one easily derives a test sequence by extracting messages on the channels from the global system state at each step. The presented determination of test sequences has been developed in analogy, as simplification and complement to the bounded model checking algorithm proposed in [4].

Translating the specification. The AutoFocus system model is translated into a Kripke structure that consists of a global state space (propositional variables), and two propositional formulas $I(s)$ and $T(s, s')$ describing the initial states of the system and its transition relation (i.e., possible successor states in the following clock cycle depending on the current state). Execution sequences of a fixed length can then be characterized with the help of these two formulas. In the following paragraphs,

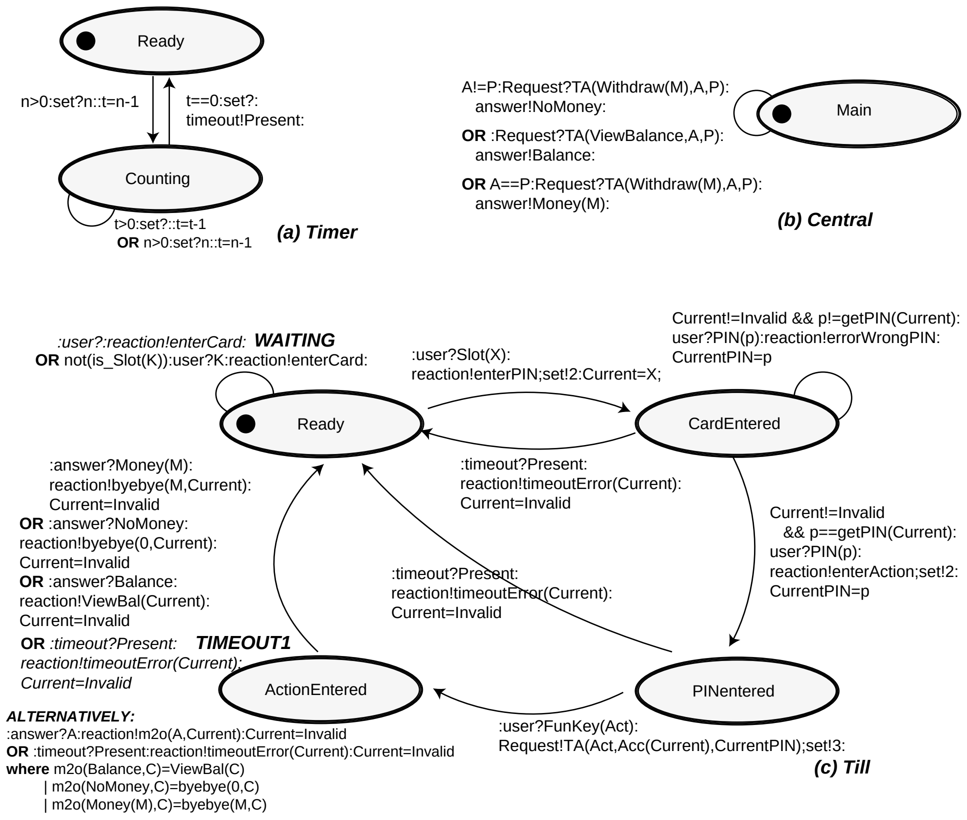

Ready
n>0:set?n::t=n-1
t==0:set?:
timeout!Present:
Counting
t>0:set?::t=t-1
OR n>0:set?n::t=n-1
(a) Timer
A!=P:Request?TA(Withdraw(M),A,P):
answer!NoMoney:
OR :Request?TA(ViewBalance,A,P):
answer!Balance:
OR A==P:Request?TA(Withdraw(M),A,P):
answer!Money(M):
Main
(b) Central
:user?:reaction!enterCard: WAITING
OR not(is_Slot(K)):user?K:reaction!enterCard:
:user?Slot(X):
reaction!enterPIN;set!2:Current=X;
Current!=Invalid && p!=getPIN(Current):
user?PIN(p):reaction!errorWrongPIN:
CurrentPIN=p
Ready
CardEntered
:timeout?Present:
reaction!timeoutError(Current):
Current=Invalid
:answer?Money(M):
reaction!byebye(M,Current):
Current=Invalid
OR :answer?NoMoney:
reaction!byebye(0,Current):
Current=Invalid
OR :answer?Balance:
reaction!ViewBal(Current):
Current=Invalid
OR :timeout?Present: TIMEOUT1
reaction!timeoutError(Current):
Current=Invalid
Current!=Invalid
&& p==getPIN(Current):
user?PIN(p):
reaction!enterAction;set!2:
CurrentPIN=p
:timeout?Present:
reaction!timeoutError(Current):
Current=Invalid
ActionEntered
PINentered
ALTERNATIVELY:
:answer?A:reaction!m2o(A,Current):Current=Invalid
OR :timeout?Present:reaction!timeoutError(Current):Current=Invalid
where m2o(Balance,C)=ViewBal(C)
| m2o(NoMoney,C)=byebye(0,C)
| m2o(Money(M),C)=byebye(M,C)
:user?FunKey(Act):
Request!TA(Act,Acc(Current),CurrentPIN);set!3:
(c) Till

Figure 2: Component Behavior

the translation of an AutoFocus system model into a Kripke structure is explained by showing how to determine its global state space and the formulas I and T for initial states and the transition relation.

Global state space. The global state space of a system specified in AutoFocus is given by the cross product of the control and data states of all its components and the current data values on the channels. Communication is modeled by treating channels as shared variables that can be written by one component and read by all connected components. AutoFocus allows for complex hierarchical data types for local variables as well as channel types. Under the assumption that the specification only uses finite data types, all these data types can be mapped to bit fields of finite length. The mapping preserves the structure of the hierarchical data types in order to increase efficiency and to keep the size of the resulting formula small when operations on these data types have to be translated into operations on bit fields. In the ATM example, the state space is (incompletely) specified by:

$$\begin{aligned}
\Sigma_{\mathsf{ATMSystem}} &= \quad \Sigma_{\mathsf{Till}} \times \Sigma_{\mathsf{Timer}} \times \Sigma_{\mathsf{Central}} \times \Sigma_{\mathsf{Channels}} \\
\Sigma_{\mathsf{Till}} &= \quad \Sigma_{\mathsf{Till.ControlState}} \times \Sigma_{\mathsf{Till.Current}} \times \Sigma_{\mathsf{Till.CurrentPIN}} \\
\Sigma_{\mathsf{Channels}} &= \quad \Sigma_{\mathsf{user}} \times \Sigma_{\mathsf{reaction}} \times \Sigma_{\mathsf{set}} \times \Sigma_{\mathsf{timeout}} \times \Sigma_{\mathsf{request}} \times \Sigma_{\mathsf{answer}} \\
\Sigma_{\mathsf{Till.ControlState}} &= \quad \mathbb{B} \times \mathbb{B} \\
\Sigma_{\mathsf{Till.Current}} &= \quad \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}
\end{aligned}$$

The state space for the control state of component Till consists of four states which can be represented by two bits. Values of the local variable Till.Current of data type Card are represented by a bit field of length five. Fig. 3 shows how the data type Card is mapped to such a bit field. The rightmost bit acts as a tag and indicates the alternative (i.e. Invalid or Valid) the current value belongs to. In case of alternative Valid, there are two arguments which prefix the tag. Note that in the general case arguments can be hierarchical data types themselves, as long as the data type definition is not recursive. Operations that manipulate hierarchical data types are then automatically converted into operations on the corresponding bit fields. Every component of $\Sigma_{\mathsf{Channels}}$ is encoded in an analogous way.

Since models have to be finite, natural numbers have to be restricted to a maximum value, maxInt. The timer's local variable t, for instance, then ranges from 0 to maxInt. For a detailed description of the translation, see [30]. For a particular

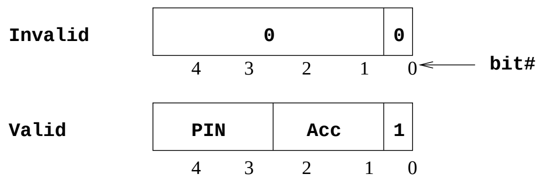


Figure 3: Encoding of data type Card (maxInt=3)

global state $s \in \Sigma_{\mathsf{ATMSystem}}$, the projection operator π is used to refer to the values of the local variables, control states and channels. $\pi_{\mathsf{Till.Current}}(s)$, for instance, refers to the five bits corresponding to the value of Till.Current, and $\pi_{\mathsf{Till.Current}_1}(s)$ refers to the second rightmost bit of this bit field.

Now $I(s)$ for the initial states and $T(s, s')$ for the transition relation can be formalized. I and T will be propositional terms, and thus have to be defined on the single bits of s or s', respectively.

Initial states. The initial state of a system specified with AutoFocus is determined by the initial control states of its automata as well as initial values for local variables, if defined. In addition, all output channels of the components must be cleared (for this purpose, there is a special value NoVal for the data type of each channel). Input channels or local variables without an initial value can have arbitrary values. For an AutoFocus system model, $I(s)$ hence consists of a conjunction of propositional terms stating that the bits of s for which an initial value is specified are set to this value.

For instance, in the ATM example the local variable Current in the Till component (storing the current card inserted into the machine) must have the initial value Invalid. In addition, the control state of this component must be Ready. As Invalid is mapped to $\mathsf{Invalid}^B = (0, 0, 0, 0, 0)$ and Ready is mapped to the bit sequence $\mathsf{Ready}^B = (0, 0)$ (superscript B denotes the Boolean representation), $I_{\mathsf{ATMSystem}}(s)$ starts with

$$I_{\mathsf{ATMSystem}}(s) = (\pi_{\mathsf{Till.Current}_0}(s) \Leftrightarrow 0) \wedge \ldots \wedge (\pi_{\mathsf{Till.Current}_4}(s) \Leftrightarrow 0) \wedge \\ (\pi_{\mathsf{Till.ControlState}_0}(s) \Leftrightarrow 0) \wedge (\pi_{\mathsf{Till.ControlState}_1}(s) \Leftrightarrow 0) \wedge \ldots$$

Transition relation for a single component. The transition relation $T(s, s')$ is given by the conjunction of the transition relations of the system components. As described above, the behavior of each component C in an AutoFocus system model is specified by a state transition diagram. With $\mathsf{Transitions(C)}$ denoting the set of component C's transitions, the idea is to translate all transitions $t \in \mathsf{Transitions}(C)$ of its STD into propositional formulas, $T_{C,t}(s, s')$. These are satisfied if the transition can fire in the current global state s, and the effect of the transition's firing is reflected by the successor state s'. $T_{C,t}$ are conjunctions of pattern matching conditions for the input statements, preconditions, output statements and postconditions.

Annotations of the AutoFocus transitions can contain user-defined functions, arithmetic operations and operations on hierarchical AutoFocus data types. [22] shows how pattern matching is performed. User-defined functions (e.g., bottom left of Fig. 2) are eliminated by replacing them with their definitions (note that this is only possible for non-recursive function definitions), and then operations on data types are replaced by predicates on the corresponding bit fields (see [30] for details). For example, the expression $a < b$ has to be translated into a predicate defined on the bits of a and b and that is satisfied if $a < b$. The translation is similar to the usual implementation in hardware [3].

The propositional formulas corresponding to the transitions can get fairly complex. One of the simplest transitions in the ATM example is the transition[1] labeled $\mathsf{WAITING}$ in Fig. 2 (c) that requests a card if there is no input from the user. It is translated as follows:

$$\begin{aligned}T_{\mathsf{Till,WAITING}}(s, s') = {} & (\pi_{\mathsf{Till.ControlState}}(s) \equiv \mathsf{Ready}^B)\wedge \\ & (\pi_{\mathsf{Till.ControlState}}(s') \equiv \mathsf{Ready}^B)\wedge \\ & (\pi_{\mathsf{Channels.user}}(s) \equiv \mathsf{NoVal}^B) \wedge (\pi_{\mathsf{Channels.reaction}}(s') \equiv \mathsf{enterCard}^B)\wedge \\ & (\pi_{\mathsf{Till.Current}}(s') \equiv \pi_{\mathsf{Till.Current}}(s)) \wedge (\pi_{\mathsf{Till.CurrentPIN}}(s') \equiv \pi_{\mathsf{Till.CurrentPIN}}(s))\wedge \\ & (\pi_{\mathsf{Channels.request}}(s') \equiv \mathsf{NoVal}^B) \wedge (\pi_{\mathsf{Channels.set}}(s') \equiv \mathsf{NoVal}^B)\end{aligned}$$

The first subterms correspond to precondition and action. $\pi_{\mathsf{Channels.reaction}}(s')$ is used rather than $\pi_{\mathsf{Channels.reaction}}(s)$ because the message written to the port can only be read in the following clock cycle (Sec. 2), and the last four subterms make sure that the local variables $\mathsf{Current}$ and $\mathsf{CurrentPIN}$ keep their values and the unused output ports are cleared. For notational convenience, $\equiv$ is used as equality on bit fields, and Ready^B, NoVal^B and $\mathsf{enterCard}^B$ are constants representing the value of the bit field this constant was mapped to (e.g., $\mathsf{Ready}^B = (0, 0)$). All subterms must still be converted to propositional formulas which is straightforward in case of the operator $\equiv$: The first subterm is translated into $(\pi_{\mathsf{Till.ControlState}^B_0}(s) \Leftrightarrow 0) \wedge (\pi_{\mathsf{Till.ControlState}^B_1}(s) \Leftrightarrow 0)$.

At each clock tick one of the transitions in an STD fires, yielding the successor state. The only exception occurs if no transition can fire. In AutoFocus system models it is usually assumed that in this case, the system remains in its current state, and output channels are cleared. This can be seen as an additional "idle transition" idle for each state that fires if the negation of all preconditions and input patterns of the other transitions is true (see [17] for details). Thus, the transition relation $T_C(s, s')$ of a component described by an STD is given by the disjunction of all $T_{C,t}$ and the idle transitions. For the ATM example,

[1]The formula only refers to the first disjunct of the transition.

$T_{\mathsf{Till}}(s,s') = \bigvee_{t \in \mathsf{Transitions(Till)}} T_{\mathsf{Till},t}(s,s') \vee T_{\mathsf{Till,idle}}(s,s')$. Note that $T_{\mathsf{Till,idle}}(s,s')$ denotes the disjunction of idle transitions for each state in Till.

Transition relation for an SSD containing multiple components. As communication between components in a system structure diagram is modeled by sharing variables for the channels, the transition relation $T(s,s')$ for a system containing multiple components is simply the conjunction of all components' transition relations: $T_{\mathsf{ATMSystem}}(s,s') = T_{\mathsf{Till}}(s,s') \wedge T_{\mathsf{Timer}}(s,s') \wedge T_{\mathsf{Central}}(s,s')$.

Unfolding the transition relation. Now that propositional formulas for I and T have been defined, all possible execution sequences of a certain fixed but arbitrary maximum length $k+1$ are given by the set of solutions to the propositional formula $\Psi = I_{\mathsf{ATMSystem}}(s_0) \wedge \bigwedge_{i=0}^{k-1} T_{\mathsf{ATMSystem}}(s_i, s_{i+1})$, a result of *unfolding* the transition relation, as explained in [4].

Ψ contains $k+1$ references s_i to the global system state, referring to the steps of the execution sequence. For $s_0, s_1, \ldots, s_k$ to be a valid execution sequence, s_0 must be an initial state, and $(s_0,s_1),(s_1,s_2),\ldots$ must be elements of the transition relation.

Determining a test sequence. Test case specifications (for example, the requirement that a certain control state be reached in one automaton, or a certain sequence of transitions be executed) are also translated automatically into predicates over control states, local variables and channels at different steps of an execution sequence (note that it is not distinguished between a predicate and its characteristic function.) This is subject of the following section. Let Φ_i denote such a predicate and Ψ the specification as a propositional formula (actually, the *bounded* model, since its maximal execution is restricted to length $k+1$). The execution sequences satisfying the test case specification are then described by the propositional formula $\Psi \wedge \Phi_i$. $\Psi \wedge \Phi_i$ is translated into conjunctive normal form, and a solution is computed by means of a standard satisfiability solver, e.g. SATO [31]. The bit fields the state space at different steps consists of are automatically mapped back to the underlying AutoFocus data types, and, by extracting the values of the message channels, the solution is converted into an I/O trace (MSC) representing the desired test sequence.

4 Testing

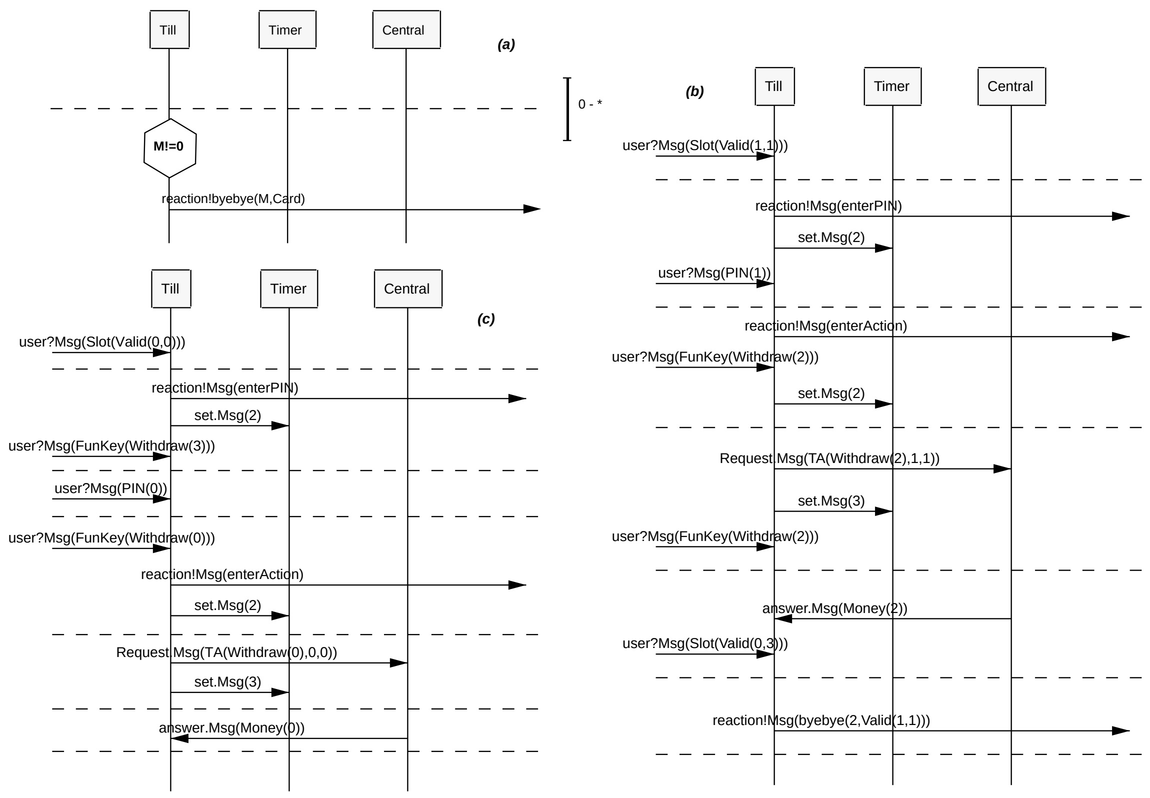
With the above translation it is possible to determine test sequences for arbitrary test case specifications given as a predicate over a finite execution sequence. In this section it is shown how system testing can be supported. To this end, three important classes of test cases are examined by referring to the ATM example. These include (partial) I/O traces, sequences of transitions, and sequences of states. Test sequences are generated fully automatically by making SATO solve $\Psi \wedge \Phi_i$ where Ψ is the formula describing the (bounded) model and Φ_i is a test case specification. Results are then translated back into I/O traces (depicted as MSCs).

Partial I/O traces. One possible test case specification is based on partially specified I/O behavior of a system. If no negations are used, such partial I/O traces can be graphically represented by MSCs. An MSC defining a use case normally does not show the complete input and output behavior of the system at each step, but only particular relevant aspects of it. By means of the translation to propositional logic and SATO's solving capabilities, an example for a complete I/O trace corresponding to the behavior described by the use case is computed. Such an I/O trace can then be used to test the implementation. It is possible to use free variables within the test case specification, bindings for which are computed by SATO and then re-translated into AutoFocus data type values. The system tester thus draws an MSC which is automatically translated into propositional logic and fed into SATO; the solver's results are then translated back into MSCs the tester can analyze.

Example. Two cases are considered: Is there a behavior where money can be withdrawn at all (Φ_1)? Is it possible to withdraw money without having entered a card (Φ_2)?

$$\Phi_1 = \bigvee_{i=0}^{k} (\pi_{\mathsf{Channels.reaction}}(s_i) \equiv \mathsf{byebye}(\mathsf{M},\mathsf{C})^B) \wedge M^B \not\equiv 0^B$$

$$\begin{aligned}\Phi_2 = \; & \bigvee_{i=0}^{k} (\bigwedge_{j=0}^{i-1} \pi_{\mathsf{Channels.user}}(s_j) \not\equiv \mathsf{Slot}(\mathsf{X})^B \wedge \\ & \pi_{\mathsf{Channels.reaction}}(s_i) \equiv \mathsf{byebye}(\mathsf{M},\mathsf{C})^B) \wedge M^B \not\equiv 0^B\end{aligned}$$



(a)
Till
Timer
Central
0 - *
M!=0
reaction!byebye(M,Card)
(b)
Till
Timer
Central
user?Msg(Slot(Valid(1,1)))
reaction!Msg(enterPIN)
set.Msg(2)
user?Msg(PIN(1))
reaction!Msg(enterAction)
user?Msg(FunKey(Withdraw(2)))
set.Msg(2)
Request.Msg(TA(Withdraw(2),1,1))
set.Msg(3)
user?Msg(FunKey(Withdraw(2)))
answer.Msg(Money(2))
user?Msg(Slot(Valid(0,3)))
reaction!Msg(byebye(2,Valid(1,1)))
(c)
Till
Timer
Central
user?Msg(Slot(Valid(0,0)))
reaction!Msg(enterPIN)
set.Msg(2)
user?Msg(FunKey(Withdraw(3)))
user?Msg(PIN(0))
user?Msg(FunKey(Withdraw(0)))
reaction!Msg(enterAction)
set.Msg(2)
Request.Msg(TA(Withdraw(0),0,0))
set.Msg(3)
answer.Msg(Money(0))

Figure 4: Test case specification (a), satisfying test sequence (b), test sequence that reaches state ActionEntered (c)

Figure 4 shows the test case specification (a) as well as a computed completed test case (I/O sequence, (b)) as an MSC. Even though the focus of this kind of application is on black box testing an implementation, it can clearly be used for white box testing a specification. Note that there are certain computed inputs that do not affect the system's behavior (e.g., the second FunKey event). This is due to the fact that input channels in reactive systems generally can carry any value. However, it is not clear if their appearance in a test case is necessarily superfluous: they may exactly lead to an unexpected behavior of the implementation.

Table 2 shows performance characteristics of the computation (with k maximum length of the test sequence, maximum integer value 5, measured on a SUN Ultra-Sparc with 1 GB memory, 400 MHz. The symbolic model checker SMV [26] takes 40.07 s for the same example.) Note that memory and time statistics exhibit a great variance (up to 500%); this is due to SATO's random choosing of the next literals to be evaluated. Shown values are averages. For the test case specification Φ_2, neither SATO (with bound up to $k = 20$) nor SMV found a test sequence, so the specified behavior is impossible.

k	CNF clauses	SATO time [s]	KByte
5	6600	0.23	225
10	13075	1.61	479
15	19550	5.63	703
20	26025	53.95	927

Table 2: Performance: Withdrawing Money (Fig. 4 (a, b))

Execution of transitions. Test cases can also be specified with the help of transitions in the automata of the AutoFocus system model. This is useful for testing if the implementation's behavior is as expected when the corresponding transitions are executed. In addition, it is possible to deploy such test cases in the white box testing process of a specification. Test cases can be specified by (possibly partial) transition sequences. One could, for instance, require that at the beginning of a system execution a certain transition sequence be fired and at some later step, another transition be fired. This technique also allows for explicitly covering certain subgraphs of an automaton. Without altering the system model, this kind of properties cannot be checked with SMV (this is why no comparative numbers are given).

Test case specifications involving transitions can also be specified using MSCs, provided they are extended by additional symbols representing the execution of a certain transition. This (and the exact semantics of such symbols) is subject of ongoing research. In addition, the tester can select transitions or transition sequences directly in AutoFocus, which provides a very intuitive and interactive way of system testing. Finally, one can also automatically compute a transition tour covering all or some of the transitions of a certain automaton using established graph algorithms (see [28]). The test sequence determination will then tell if the transition tour is executable, and if so, will give as a result a corresponding I/O behavior of the system.

Example. Figure 5 (c) shows a computed test sequence for the (graphical) specification "transition TIMEOUT1 must be fired at some step of the execution" ($\Phi_3 = \bigvee_{i=0}^{k-1} T_{\mathsf{Till,TIMEOUT1}}(s_i, s_{i+1})$), It also exhibits two test cases for two complete transition cycles (full transition coverage) with (a) and without (b) transition TIMEOUT1. This is a good example for how testing helps detecting errors in a specification. When trying to find a sequence that covers all transitions, one quickly discovers that TIMEOUT1 should never fire[2] for the central database immediately reacts once it received a request. However, if solely transition TIMEOUT1 is to be tested, the system discovers a run where TIMEOUT1 does indeed fire (Fig. 5 (c)): The timer is set to 2 during transition to state PINEntered. Two ticks later, the user requests an action (ViewBalance). As a result, the transition to ActionEntered is taken, but at the same time, a timeout occurs, just before the timer is reset to 3. This causes TIMEOUT1 to fire. The problem is that the system model simply is wrong (simple modifications to the model could avoid such a case). This kind of "race conditions" often occur because modelers do not pay enough attention – in this case, the problem is due to the fact that the presented model is an abstraction of a much more complicated system [30], and the abstraction was not chosen carefully enough. Table 3 shows performance data for the transition tour (length 18) without TIMEOUT1 for different maximal integer values. Table 4 shows performance characteristics for computation of a test case where TIMEOUT1 is eventually fired.

Note that there is room for optimization. As described in chapter 3, the formula Ψ used to model all possible bounded executions of the system contains the disjunction of subterms for all transitions in the components. To find I/O data for a transition sequence, in each step of unfolding all transitions of the component that do not occur in the specified transition sequence can be omitted ("slicing the model"). This results in considerable performance gains (compare the performance data for the 18-step transition tour in table 3 with the results in table 2). This optimization is performed automatically and is inherent to the translation scheme.

maxint	CNF clauses	SATO time [s]	KByte
3	5671	0.25	224
5	7214	0.25	289
15	9833	0.25	383
63	14508	0.27	543
1023	25166	0.53	895
8191	27615	0.55	960

Table 3: Performance: Transition tour without TIMEOUT1 (Fig. 5 (a))

Reachability of states. As a last example, one can specify particular control states (or sequences thereof) to be reached. Applications include putting a system into an error state (rather than observing the transitions' outputs), to check reachability conditions, or to perform consistency checks ("is it possible to enter state ActionEntered without having entered a card?"[3]). Such test cases can again be spec-

[2]For reasons of clarity, transitions between the same states have been connected by OR in Figure 2. In AutoFocus, they are represented as different transitions, i.e., TIMEOUT1 is the last of the four transitions leading from ActionEntered to Ready.

[3]$\Phi'_4 = \bigvee_{i=0}^{k} (\bigwedge_{j=0}^{i-1} \pi_{\mathsf{Channels.user}}(s_i) \not\equiv \mathsf{Slot(X)}^B \wedge \pi_{\mathsf{Till.State}}(s_i) \equiv \mathsf{ActionEntered}^B)$

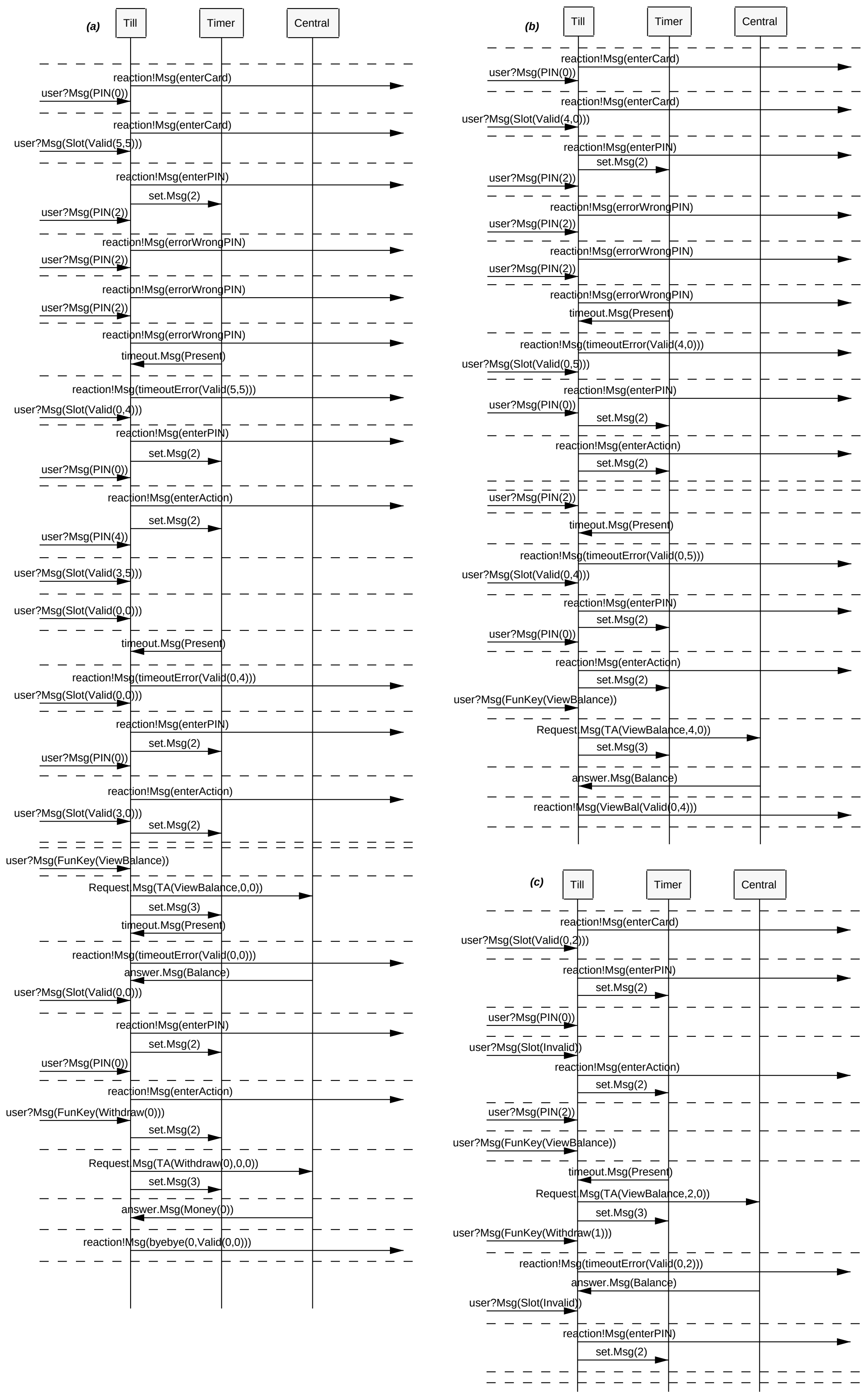

(a)
Till
Timer
Central
reaction!Msg(enterCard)
user?Msg(PIN(0))
reaction!Msg(enterCard)
user?Msg(Slot(Valid(5,5)))
reaction!Msg(enterPIN)
set.Msg(2)
user?Msg(PIN(2))
reaction!Msg(errorWrongPIN)
user?Msg(PIN(2))
reaction!Msg(errorWrongPIN)
user?Msg(PIN(2))
reaction!Msg(errorWrongPIN)
timeout.Msg(Present)
reaction!Msg(timeoutError(Valid(5,5)))
user?Msg(Slot(Valid(0,4)))
reaction!Msg(enterPIN)
set.Msg(2)
user?Msg(PIN(0))
reaction!Msg(enterAction)
set.Msg(2)
user?Msg(PIN(4))
user?Msg(Slot(Valid(3,5)))
user?Msg(Slot(Valid(0,0)))
timeout.Msg(Present)
reaction!Msg(timeoutError(Valid(0,4)))
user?Msg(Slot(Valid(0,0)))
reaction!Msg(enterPIN)
set.Msg(2)
user?Msg(PIN(0))
reaction!Msg(enterAction)
user?Msg(Slot(Valid(3,0)))
set.Msg(2)
user?Msg(FunKey(ViewBalance))
Request!Msg(TA(ViewBalance,0,0))
set.Msg(3)
timeout.Msg(Present)
reaction!Msg(timeoutError(Valid(0,0)))
answer.Msg(Balance)
user?Msg(Slot(Valid(0,0)))
reaction!Msg(enterPIN)
set.Msg(2)
user?Msg(PIN(0))
reaction!Msg(enterAction)
user?Msg(FunKey(Withdraw(0)))
set.Msg(2)
Request!Msg(TA(Withdraw(0),0,0))
set.Msg(3)
answer.Msg(Money(0))
reaction!Msg(byebye(0,Valid(0,0)))
(b)
Till
Timer
Central
reaction!Msg(enterCard)
user?Msg(PIN(0))
reaction!Msg(enterCard)
user?Msg(Slot(Valid(4,0)))
reaction!Msg(enterPIN)
set.Msg(2)
user?Msg(PIN(2))
reaction!Msg(errorWrongPIN)
user?Msg(PIN(2))
reaction!Msg(errorWrongPIN)
user?Msg(PIN(2))
reaction!Msg(errorWrongPIN)
timeout.Msg(Present)
reaction!Msg(timeoutError(Valid(4,0)))
user?Msg(Slot(Valid(0,5)))
reaction!Msg(enterPIN)
user?Msg(PIN(0))
set.Msg(2)
reaction!Msg(enterAction)
set.Msg(2)
user?Msg(PIN(2))
timeout.Msg(Present)
reaction!Msg(timeoutError(Valid(0,5)))
user?Msg(Slot(Valid(0,4)))
reaction!Msg(enterPIN)
set.Msg(2)
user?Msg(PIN(0))
reaction!Msg(enterAction)
set.Msg(2)
user?Msg(FunKey(ViewBalance))
Request!Msg(TA(ViewBalance,4,0))
set.Msg(3)
answer.Msg(Balance)
reaction!Msg(ViewBal(Valid(0,4)))
(c)
Till
Timer
Central
reaction!Msg(enterCard)
user?Msg(Slot(Valid(0,2)))
reaction!Msg(enterPIN)
set.Msg(2)
user?Msg(PIN(0))
user?Msg(Slot(Invalid))
reaction!Msg(enterAction)
set.Msg(2)
user?Msg(PIN(2))
user?Msg(FunKey(ViewBalance))
timeout.Msg(Present)
Request!Msg(TA(ViewBalance,2,0))
set.Msg(3)
user?Msg(FunKey(Withdraw(1)))
reaction!Msg(timeoutError(Valid(0,2)))
answer.Msg(Balance)
user?Msg(Slot(Invalid))
reaction!Msg(enterPIN)
set.Msg(2)

Figure 5: Executions with full coverage with (a) and without (b) transition TIMEOUT1; Execution with transition TIMEOUT1 (c)

k	CNF clauses	SATO time [s]	KByte
5	6558	0.23	255
10	12998	0.45	479
15	19438	4.64	703
20	25878	37.26	895

Table 4: Performance: Transition TIMEOUT1 (Fig. 5 (c))

ified using MSCs (with the help of condition boxes) or graphically by clicking on the states.

Example. Figure 4 (c) shows a test case for the specification that requires the system to reach state ActionEntered ($\Phi_4 = \bigvee_{i=0}^{k}(\pi_{\mathsf{Till.ControlState}}(s_i) \equiv \mathsf{ActionEntered}^B)$).

Remarks. (1) Obviously, problems occur if the system model is nondeterministic, and a computed I/O trace does not correspond to the implementation's actual behavior even though it is correct. However, a restriction to deterministic models seems suitable since such models are usually easier to understand.

(2) There is a close relationship to bounded model checking [4]. Bounded model checking restricts model checking to executions of a certain maximum length and results in remarkable efficiency gains over classical model checking. However, unlike counter examples for general LTL formulas, which can contain loops, test sequences are always finite. Therefore, for the purpose of determining a test sequence for a given test case specification, the mere unfolding of the transition relation to the intended length of the test sequence is sufficient. In addition, as test sequences are finite, test cases can always be specified using predicates over state variables (control states, local variables, and channels) at the different execution steps ($0..k$). Since there is no need for an elaborate translation of general LTL specifications and infinite execution sequences into propositional formulas as in bounded model checking, these formulas become much simpler and thus test sequences can be determined more efficiently. Furthermore, without altering system models, LTL does not allow for explicit statements about transitions. This does not mean that LTL formulas are not appropriate for testing purposes. However, they may increase complexity and are not as easily understood by the tester as graphical specifications.

(3) The translation of AutoFocus system models and test case specifications into propositional formulas is fully automatic and integrated into the AutoFocus CASE tool. Test cases can be specified in the tool by MSCs representing partial I/O traces, or by selecting transitions or control states (or sequences thereof). The computed test sequences are immediately translated back into MSCs that can be displayed graphically.

(4) The given translation of AutoFocus system models into propositional logic can also be used for classical symbolic model checking: a Kripke structure consisting of a boolean state space and the propositional formulas $I(s)$ and $T(s, s')$, are exactly the input for a symbolic model checking algorithm. Actually, AutoFocus supports SMV model checking, bounded model checking and test sequence generation using SATO — all based on the same translation. The only overhead involved by translating the specification into the SATO input format is that the transition

relation must be unfolded and then converted into conjunctive normal form, which is negligible considering the complexity of the SAT solving algorithm.

5 Related Work

This section sets the presented work in context with other approaches such as model checking, bounded model checking, and techniques based on a constrained enumeration of test sequences. In addition, the use of MSCs for the specification of test cases is discussed. Finally, some relevant references are cited.

Generating test sequences. The use of propositionally encoded specifications with a certain bound is similar to bounded model checking, but the method presented above exhibits some differences. Its intention is not a full coverage of a linear logic (e.g., LTL) but rather a subset that turns out to be (a) conveniently representable by MSCs and (b) more efficiently solvable. In addition, firing of transitions can be expressed, which is impossible using LTL formulas without altering the model. As it is related to bounded model checking, determining test sequences using SATO is more efficient than determining test sequences using SMV for the same reasons for which bounded model checking is in some cases superior to symbolic model checking using BDDs (see [4]): building the BDDs representing the system model is expensive as the size of the BDD corresponding to a formula is dependent on the size of the BDDs corresponding to its subformulas – whereas a system representation as a propositional formula can always be determined in polynomial time. However, the advantage of bounded model checking (and thus of the presented approach) diminishes when longer execution sequences are to be considered, as the complexity of the Davis-Putnam-algorithm used by SATO is exponential in the size of the formula.

The knowledge of the structure of the system (communicating automata) also allows for optimizations such as model slicing: for instance, when computing input data for a certain transition tour, possible traces of the model are restricted to this very transition tour. This yields considerable performance gains, as explained in section 4.

The possibility to use counter examples from model checkers for testing purposes is widely recognized. In Uppaal [16], for instance, these traces can be fed into the tool's simulation environment and used for debugging. The difference with the approach presented in this paper is the use of a full model checker with its inherent performance problems.

Other approaches to validating systems (computing test sequences) include model checking-on-the-fly (e.g., [21, 8]) or constraint solving techniques [6, 17, 18]. In contrast to (unbounded) model checking, they share the commonality of not building the entire state space before checking whether or not a property holds. Instead, the search tree is pruned in an ad-hoc (on-the-fly) or a-priori manner (constraints). Model checking-on-the-fly suffers from a lack of efficient representations such as BDDs (but is, in some cases, more efficient). Constraint solving techniques are heavily dependent on the availability of efficient constraint solvers (which, on the basis of Constraint Handling Rules in Constraint Logic Programming, are the subject of ongoing work [17]). These approaches are rather explorative. Unless one considers the mere presence of backtracking in resolution procedures as an indication for their explorative nature, the presented approach is not: Its idea consists of conjugating a property with the (possibly abstracted) system and to find a solution to this formula. It is thus fundamentally different, even though determining the satisfiability of propositional formulae surely is a kind of constraint solving. Constraint solving techniques (over domains other than the Booleans) surely are a promising approach to system validation [18].

In this context, another approach is worth mentioning. It consists of using automata that encode properties to be tested (e.g., [16]). These automata are run in parallel with the system under test, and they enter an error state when the property is violated. They are hence similar to model checking-on-the-fly. Note that the automata are actually run and not, as in the case of the automata-theoretic interpretation of model checking, intersected after complementation of one of them and checked for emptiness [29].

Specifying test cases. In terms of representation, automata are a good choice when general properties are to be tested. However, when sequences of transitions, sequences of states or component interactions, i.e., partial I/O traces, form the major part of the property to be tested, they seem to be more complicated than MSCs. Typical test purposes include bounded liveness properties that require a notion of final states in (Büchi) automata which renders them more complex. Particularly for the first kind of test case specifications, Sequence Charts seem to be more appropriate. Note that this argumentation is about (intuitive) representation rather than expressivity. This motivates the following discussion on specification languages for test cases.

MSCs explicitly visualize the progress of time in a sequential manner and allow for a simple specification of bounded liveness properties (for they are bounded by definition). They describe exemplary component interactions, and are usually deployed to specify use cases that show a certain required behavior of the system model or its implementation, respectively. Condition boxes in MSCs can be used to specify system states [15, 11] - a fact that is used in work on their translation into automata [15]. The incorporation of special symbols for transitions into MSCs is the subject of ongoing work. Note that this does not advocate the integration of MSCs and automata into a new kind of language for the specification of systems, but rather for the specification of test cases. In terms of system specification, a clear distinction between behavior and interaction view seems to be reasonable.

The specification of properties also necessitates a construct for negation. If negation is to be used on levels of the property other than the outmost one (i.e., "the following property should *not* hold"), there is a need for describing "partial" negation within MSCs which is the subject of ongoing work. In addition, for the specification of test cases, the specification of iteration is needed within MSCs (e.g., Fig. 4).

One problem with these extensions is the formal definition of their semantics which, in case of negation, turns out to be far from being trivial. Another complication arises from the fact that there are different interpretations of MSCs. Two subsequent messages (arrows) in a diagram may mean "nothing but these two messages occur within the specified time in the specified order", or they may mean "the partial I/O behavior in question must contain the two specified messages in their specified order". It seems to be reasonable to admit both interpretations.

Summarizing, the authors consider MSCs as an appropriate, intuitive means for test case specifications in cases where I/O behavior is to be tested. This necessitates constructs for iteration (e.g., Fig. 4). Furthermore, for sequences of states or transitions, MSCs seem to be a good choice, provided that constructs for states (condition boxes) or transitions (special arrows) are supported.

The presented approach to compute test sequences actually uses a more general form of specifying test cases, a predicate defined over the steps of a finite execution sequence. MSCs can easily be translated into such a predicate, but so can other ways of specifying test case, like selecting states or transitions interactively in the tool.

General comments. The derivation of test cases is a particularly difficult and challenging problem, and there are many approaches to solving it. The following brief summary is necessarily incomplete; no work on regression testing, integration testing, test management, or hardware testing is cited.

The classification tree method [10] offers tool support, helps in manually designing test cases and is applied in industry. [24], for instance, explores automatic test case generation with Z and this method that might also be applied to the approach presented in this paper. [20] discusses criteria for the generation of test cases. [1] describes assessment techniques for specification based testing. By their very nature, both references are applicable to all methods of generating test sequences. There are several formalizations of testing: for (extended) finite state [28, 23], for algebraic specifications [7], and for general labeled transition systems [5, 27].

MSCs as a language for test case specifications are described in [9]. The focus is on telecommunication. [17, 6] advocate the use of Constraint Logic Programming for software validation for reactive systems. [19] may serve as an example for the use of Prolog for testing purposes, based on the structured object-oriented formal language (SOFL). [2] uses counterexamples of a model checker in combination with abstractions and mutation analysis as the basis for the generation of test cases. Their approach is different from the presented one in that there is no use of specialized solvers for a special class of test cases. Furthermore, the analysis of mutations is not part of the presented work. [4] contains many ideas that have been adopted to the above translation of specifications into propositional logic. Finally, [30] describes many aspects of this work in more detail.

6 Conclusion

In this paper, an approach to automatically determining test sequences from a finite system specification by translating an AutoFocus model and test case specifications into propositional logic has been presented. AutoFocus supports graphical system models, a simple semantics that does not lead to an explosion of generated formulas, and can therefore be used to formulate test cases in an appropriate way. The presented method provides support for both white box testing strategies on AutoFocus models and the derivation of correct test cases for arbitrary implementations.

Test designers do not need to deal with specifications in temporal logic that are hard to understand. It suffices to point out interesting test cases in terms of Au-

AutoFocus' graphical description techniques (by the use of MSCs, state definitions, or by clicking on transitions). This is possible because of the restriction to test case specifications that include partial I/O traces, sequences of transitions, and sequences of states. The use of the presented method seems particularly suited to handle these classes (for they do not interfere right from the beginning with the idea of just regarding system runs of a fixed length - the validity of safety properties, for instance, may be approximated with this method, but it is counterintuitive to check them only for short finite runs). It is not intended to test arbitrary formulae specified in a temporal logics. In particular, the approach will exhibit problems with large-scale problems and long test sequences (20 steps are only realistic for either simple or very abstract system specifications) for the same reasons as other approaches: state space explosion. Classical abstraction techniques such as more elaborate slicing or the abstraction of integer values into a small number of "equivalence" classes can alleviate this problem. This is the subject of ongoing work.

The presented method does not support recursive data type or function definitions (on transitions; see [17] for a possible solution on the basis of Constraint Logic Programming). Obviously there is a strong need for further evaluation of the produced test cases. Just a few examples have been presented, but the full strength of the method must be evaluated with an experiment of bigger size and not necessarily correct implementations.

Another improvement in the testing process is the use of an appropriate graphical language for test case specifications. MSCs seem to be a reasonable choice [9], notwithstanding some modifications are desirable, e.g., expressing the absence of particular signals (negated transitions). In addition, the interactive graphical specification of test cases (e.g. by clicking on states or transitions) helps in debugging the system model. The current implementation does not support general condition boxes for the specification of test cases. Besides, the implementation only supports computation of one single test sequence. It might be desirable to get all or some other test sequences – only a slight modification to the implementation. Future work also includes the handling of hierarchical STDs, the determination of other typical relevant test case specifications (e.g., reachability analyses for optimization), and an assessment of the created test sequences by means of known or new metrics.

Acknowledgment. The authors would like to thank Alexander B. Schmidt and Thomas Stauner as well as the anonymous referees for helpful comments on this paper.

References

[1] S. Allen and M. Woodward. Assessing the Quality of Specification-based Testing. In S. Bologna and G.Bucci, editors, *Proc. 3rd Intl. Conf. on Achieving Quality in Software (AQuIS '96)*, pages 341–354, Florence, Italy, 1996.

[2] P. Ammann and P. Black. Abstracting Formal Specifications to Generate Software Tests via Model Checking. In *Proc. 18th Digital Avionics Systems Conference (DASC'99)*, volume 2, pages 10.A.6.1–10, St. Louis, MO, October 1999. IEEE.

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

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In W. Cleaveland, editor, *Proc. TACAS/ETAPS'99*, LNAI 1249, pages 193–207, 1999.

[5] E. Brinksma. A theory for the derivation of tests. In S. Aggarwal and K. Sabnani, editors, *Proc. 8th Intl. Conf. on Protocol Specification, Testing, and Verification*, pages 63–74, 1988.

[6] A. Ciarlini and T. Frühwirth. Using Constraint Logic Programming for Software Validation. In *5th workshop on the German-Brazilian Bilateral Programme for Scientific and Technological Cooperation*, Königswinter, Germany, March 1999.

[7] M. Gaudel. Testing can be formal, too. In P. Mosses, M. Nielsen, and M. Schwartzbach, editors, *Proc. Intl. Conf. on Theory and Practice of Software Development (TAPSOFT'95)*, LNCS 915, pages 82–96, Aarhus, Denmark, May 1995.

[8] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. 15th Workshop on Protocol Specification, Testing, and Verification*, Warsaw, June 1995.

[9] J. Grabowski. *Test Case Generation and Test Case Specification with Message Sequence Charts*. PhD thesis, Universität Bern, 1994.

[10] M. Grochtmann and K.Grimm. Classification trees for partition testing. *Software Testing, Verification, and Reliability*, 3:63–82, 1993.

[11] R. Grosu, I. Krüger, and T. Stauner. Hybrid Sequence Charts. In *Proc. 3rd IEEE Intl. Symp. on Object-oriented Real-time distributed Computing (ISORC 2000)*. IEEE, 2000.

[12] F. Huber, S. Molterer, A. Rausch, B. Schätz, M. Sihling, and O. Slotosch. Tool supported specification and simulation of distributed systems. In B. Krämer, N. Uchihira, P. Croll, and S. Russo, editors, *Proc. Intl. Symp. on Software Engineering for Parallel and Distributed Systems*, pages 155–164. IEEE, 1998.

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

[14] ITU. ITU-T Recommendation Z.120: Message Sequence Charts (MSC), November 1999.

[15] I. Krüger. *Distributed Systems Design with Message Sequence Charts*. PhD thesis, Munich University of Technology, 2000.

[16] K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Springer International Journal of Software Tools for Technology Transfer*, 1(1+2), 1997.

[17] H. Lötzbeyer and A. Pretschner. AutoFocus on Constraint Logic Programming. In *Proc. (Constraint) Logic Programming and Software Engineering (LPSE'2000)*, London, July 2000.

[18] H. Lötzbeyer and A. Pretschner. Testing Concurrent Reactive Systems with Constraint Logic Programming. In *Proc. 2nd workshop on Rule-Based Constraint Reasoning and Programming*, Singapore, September 2000.

[19] J. Offutt and S. Liu. Generating test data from SOFL specifications. *J. of Systems and Software*, 49(1):49–62, December 1999.

[20] J. Offutt, Y. Xiong, and S. Liu. Criteria for generating specification-based tests. In *Proc. 5th IEEE Intl. Conf. on Engineering of Complex Computer Systems (ICECCS '99)*, Las Vegas, USA, October 1999.

[21] D. Peled and W. Penczek. Using Asynchronous Büchi Automata for Efficient Model-Checking of Concurrent Systems. In *Proc. 15th Workshop on Protocol Specification, Testing, and Verification*, Warsaw, June 1995.

[22] J. Philipps and O. Slotosch. The quest for correct systems: Model checking of diagrams and datatypes. In *Proc. IEEE Asian Pacific Software Engineering Conf. (APSEC'99)*, pages 449–458, 1999.

[23] S. Sadeghipour and H. Singh. Test strategies on the basis of extended finite state machines, 1998. Report FT3/SM-98-04, Daimler-Benz AG.

[24] H. Singh, M. Conrad, and S. Sadeghipour. Test case design based on Z and the Classification-Tree Method. In *Proc. Workshop "Tool Support for System Development and Verification*, Bremen, Germany, 1996.

[25] 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.

[26] SMV. http://www.cs.cmu.edu/~modelcheck/.

[27] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software–Concepts and Tools*, 17(3):103–120, 1996.

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

[29] M. Vardi and P. Wolter. An automata-theoretic approach to automatic program verification. In *Proc. 1st IEEE Symp. on Logic in Computer science*, pages 332–334, boston, 1986.

[30] G. Wimmel. Specification Based Determination of Test Sequences in Embedded Systems. Master's thesis, Technische Universität München, 2000.

[31] H. Zhang. SATO: An efficient propositional prover. In W. McCune, editor, *Proc. 14th Intl. Conf. on Automated deduction*, LNAI 1249, pages 272–275. Springer, July 13–17 1997.