Proc. 6th Intl. Workshop on Formal Methods for Industrial Critical Systems, 7/01, pp.79-94

Model Based Testing for Real: The Inhouse Card Case Study

A. Pretschner[1], O. Slotosch[2], H. Lötzbeyer[1], E. Aiglstorfer[3], S. Kriebel[3]

[1] *Institut für Informatik, Technische Universität München*
{pretschn,loetzbey}@in.tum.de

[2] *Validas Model Validation AG*
slotosch@validas.de

[3] *Giesecke&Devrient GmbH*
{Ernst.Aiglstorfer,Stefan.Kriebel}@gdm.de

Abstract

We describe the modeling concepts of the CASE tool AutoFocus as well as our Constraint Logic Programming based approach to model-based test case generation along the lines of an inhouse smart card case study. Besides testing the model itself, we used the generated test cases to validate the respective properties in the actual hardware.
Keywords. Automatic test case generation, CASE, reactive systems, validation.

1 Introduction

This paper summarizes the results of a feasibility study that was carried out by TU München, Validas Model Validation AG, and Giesecke&Devrient GmbH. Its purpose was to determine the industrial applicability of a combination of the CASE tool AutoFocus [18] with a prototype for the automatic generation of test cases on the grounds of Constraint Logic Programming (CLP) [21]. In this article, the application domain is that of smart cards. Generating test cases for these systems turns out to be a tedious and difficult task, and the potentials of automatization were to be assessed. The main result of this study is that for this application domain, the techniques in question are convincing candidates for further exploration, as expressed in concrete plans for further collaboration of the three partners.

Overview. The paper is organized as follows. In Section 2, we briefly present the CASE tool AutoFocus for specification, simulation, and validation of reactive systems. Section 3 then describes the case study. In section 4, we describe and discuss our approach to the generation of test cases as well as its embedding into an incremental development process, and present some experimental results. The paper concludes with an assessment of the results of this feasibility study which takes into account both the modeling formalisms of AutoFocus and the generation of test cases.

Related Work. A theory of formal testing is tackled in [16, 4]. They share the commonality of defining an observational congruence ("selection hypotheses") on systems. Sim-

ilar relations are used in [31, 30] to compute whether or not a system (model) conforms to its specification. We differ from this approach in that we do not want to prove such a conformance relation but rather approximate its proof as done in traditional testing (without an explicit formalization of the conformance relation). (Constraint) Logic Programming for test case generation has been used in [24, 7, 23]; our approach differs (1) in the class of systems we consider, (2) in the input language with a concept of interface and a combined approach to behavior specifications with automata and functional definitions on transitions, and (3) in the thereby induced necessity for powerful, yet existing, constraint handlers on the grounds of Constraint Handling Rules (CHR, [14]). Lutess [11] is a tool for the generation of test cases for Lustre (as is Gatel [23], see above). The difference with our approach is the use of model checkers or random number generators for the generation of test cases as well as a restriction to boolean data types. Code generation on the grounds of CLP is, for various non-modular [25] automata considered in [17, 13].

The relationship of Model Checking and (C)LP with possibly tabled resolution procedures is discussed (and used) in [9, 12, 8]. Our approach as a mixture between enumerative (explicit) and symbolic approaches to (bounded) model checking. Its symbolic nature is characterized by the fact that constraints allow for storing possibly infinite sets of states or inputs. Its explicit nature stems from the search algorithms (depth-first, breadth-first, best-first, tabu) we deploy. Our approach is not directly applicable to proving general properties; the properties we are interested in are usually existentially path-quantified (properties that, in general, are proved by referring to universally path-quantified formulas by means of "shifting" negations). These issues are discussed in more detail in [27, 26]. The extension of our approach to symbolic on-the-fly model checking is the subject of current work.

In the context of mutation testing, constraints for the generation of test cases for transformational systems are used in [10]. The idea is to formulate constraints that approximate criteria for killing mutants.

[6] uses a mixture of BDDs and Presburger constraints for the representation of sets of states in reactive systems. [1] uses linear constraints on real numbers for model checking hybrid systems. Clearly, the focus is on model checking. The difference with our approach is that (1) we are mixing enumerative and symbolic techniques rather than computing fixed points on sets of constraints and (2), again, use CHR with constraint solvers on arbitrary domains (e.g., FD) for allowing convenient interactions and user-defined specifications of test cases.

Evolutionary approaches to test case generation (e.g., [**?**]) are similar to our approach w.r.t. the importance of finding good fitness functions for the search strategies that are used. Usually, these approaches are concerned with transformational systems, and the technique of generation test cases (modification of random tests) differs from ours.

For the sake of simplicity, we synonymously speak of test cases and test sequences as sequences of I/O traces of a system. [22] contains a more precise terminology.

2 AutoFocus

AutoFocus (`autofocus.in.tum.de`, [18]) is a tool for the graphical specification and validation of reactive (embedded) systems. In terms of its focus on behavior models (automata), it is quite similar to a subset of the UML-RT, but we consider its simple and formalized semantics to be a prerequisite for validation techniques such as model checking or testing.

Components. Systems are structured by decomposing them into *components*. A component represents a single unit of computation. Components synchronously communicate via typed and directed *channels*. Each end of a channel is connected to a *port*. Ports belong to a component, and since channels are directed, the same holds true for ports, and the ports of a component constitute hence its interface. When two components are connected by a channel, we say that they are composed.

Composition of components is depicted in *System Structure Diagrams* (SSDs). SSDs may be hierarchical; boxes represent components, and arrows between them represent channels. Our case study merely consists of a single non-hierarchic component that is depicted in Fig. 1 (the complexity of our example system lies in the behavior, not in the structure). Components may be associated with a set of local variables that are manipulated by the component's behavior; these local variables form the component's *data space*.

Behavior. Bottom level components, i.e., components that are not composed of other components, are equipped with a behavior. Behaviors are specified by means of extended state machines: finite state machines that can access input and output ports as well as local variables of the (bottom level) component they belong to. Fig. 4 shows the pictorial representation of such an automaton, a *State Transition Diagram* (STD). Circles represent control states, and arrows between them represent transitions. Transitions may *fire* if (1) certain pattern matching (PM) conditions of the form[1] *channel?pattern* on the input channels hold and (2) their guard (G) holds. Firing then means to update local variables (PC_V) and write outputs (PC_O); this is performed in the so-called postconditions. Postconditions are thus nothing but assignments of local variables or output channels. Each transition is hence associated with a quadruple $G : PM : PC_V : PC_O$ where some parts may be omitted: lack of a guard, for instance, means that the condition may always fire, provided its input pattern matching condition is satisfied. Note that in Fig. 3 transitions are simply labeled since the respective quadruples would clutter the diagram.

Data and Functions. Channels were said to be typed. Types include standard types like integers or booleans, but they may also be user-defined in a Gofer-like functional language. This enables one to concisely describe enumeration or inductive types. The same functional language may be used for the definition of new, possibly recursive, functions. These functions can then be used in the guard or postcondition of a transition.

Execution. AUTOFOCUS components execute concurrently and simultaneously in a time synchronous manner. The existence of a global clock ensures existence of so-called *ticks*. Before each tick, every component reads it input ports. It then computes pattern matching conditions and preconditions for all possible transitions. Possibly non-deterministically, one of them is then chosen; if there is none, the system idles. In addition, it computes new values for local variables and output ports. During the (instantaneous) tick, it updates the respective variable, and writes new values to the output ports. Since channels connect output to input ports and transferring messages does not consume any time, these values are immediately available for the connected component after the tick. The procedure then repeats. This results in a time-synchronous communication scheme with buffer size 1.

[1] *pattern* may well be the empty string; this means that no input is allowed at *channel*.

Interaction. Besides the architecture (SSD), behavior (STD), and data views, the interaction view plays an important role in coping with reactive systems. Sequence diagrams play thus an integral part in the specification of systems, test cases, for simulation results, and for requirements capture. Appropriate editors have been connected to AutoFocus.

Validation. Besides the modeling capabilities of AutoFocus, there are several tools for validating the specified models. These include simulators on the grounds of code generators for languages such as C, Java, Prolog, or ADA. Furthermore, model checkers (SMV, μcke), propositional solvers for test case generation (SATO), and theorem provers (VSE) have been connected to the tool. Currently, test tools such as ATTOL coverage/unit test or DOORS for requirements tracing are connected to AutoFocus.

3 The Inhouse Card

In this section we describe two equivalent models of the inhouse card, a sample study of Giesecke & Devrient. The purpose of this smart card is to serve as a security token for personal access control, e.g., to various areas of a company site, or personalized computer access. One model has been built from the modeler's point of view, and the other one from the application designer's view. We focus on the application view and show the differences to the other model.

The inhouse card is a secure device which allows the storage of secret keys. When the smart card comes into contact with a terminal, authentication protocols are run to start the communication. Encryption and secret keys ensure secure communication. The application programming interface for the card conforms to ISO 7816 [19] which comprises the relevant commands for authentication (smart card - terminal) and verification (user - terminal / user - smart card) routines. For the authentication process symmetric cryptography with secret keys is used. The verification process is based on the known personal identification number (PIN) handling. The number of failings is limited by counters. The model describes the behavior of the card, and it is used to determine test sequences. Test sequences shall have a good coverage among the available commands.

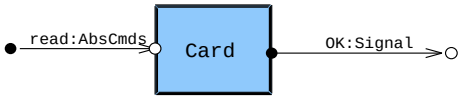

Figure 1: Interface of the Inhouse Card

3.1 Interfaces

Both models have the same interface and structure (see the SSD in Figure 1). The structure shows the interfaces of the modeled component `Card`. It receives commands from the environment via channel `read`, and sends return values via channel `OK`. The commands of the inhouse card and their return values are described in the specification manual. In the following, however, we restrict ourselves to giving an abstracted version of the actual card which we could, nonetheless, use for the computation of actual test cases. After appropriate transformations, these were fed into actual hardware, i.e., in this case study, we did not test the model but rather the implementation.

For instance, we used the following equivalence classes for the `verify` command, an integral part of the verification protocol:

```
data AbsCmds =
    Verify_A | // correct state & PIN
    Verify_B | // wrong parameters
```

```
Verify_C | // wrong state
Verify_D;  // wrong PIN
```

This command is sent from the reading device to the card. These equivalence classes encode not only the command itself as well as its parameters, but also the return value from the card: `Verify_A` denotes sending the command with correct parameters and return value `OK`.

However, since in AutoFocus all elements of an input type can be entered at any time, it is necessary to differentiate between allowed cases and impossible cases. For instance, since a successful verify command cannot be entered in every state, the equivalence classes `Verify_A` and `Verify_C` are disjoint, i.e., they cannot be tested in the same states. Our modeling task included hence the exclusion of some commands in some states. Therefore, the model sends signals of the single valued type `data Signal = Present` to the environment to indicate that the received command has been admissible. For the generation of test sequences we are only interested in admissible sequences, i.e., in sequences that accept every command with a signal `Present` on the return channel `OK`.

3.2 Behavior

As mentioned above, the inhouse card is used for access control. The user puts the card into a card reader, and with a correct PIN, he can access the respective part of the building. The card has also a super-user mode (with a Personal Unblocking Key, or PUK). Authentication is achieved between card reader and (super-user) terminals. Authentication is based on encryption of random numbers (so-called challenges). All (different) authentications follow the same scheme. Using our command equivalence classes, the simplified sequence is depicted in Figure 2.

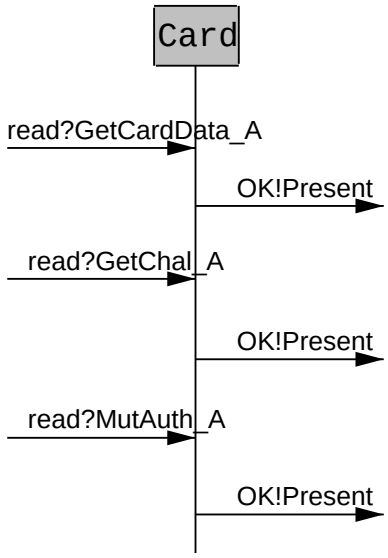

Figure 2: Authentication Sequence (simplified)

There are six counters that count the number of authentication attempts (for different situations). These counters `K1C:Int, ..., K6C:Int` are declared as local variables of component `Card`. Different maximum values are declared for these counters as constants to model the fact that the PIN can be entered three times wrongly, and the PUK 14 times before they are blocked (the maximum value for the first counter, for instance, is declared by `const startK1 = 14`).

The description of the behavior consists of several main states (called "authentication" states in the requirements specification), and transitions between them. Some transitions (for instance, the reset transition) change the main state directly, whereas the authentication process requires two intermediate states between the connected authentication states.

There are two ways of modeling these intermediate states in AutoFocus: The modeler's view is to model the intermediate states as control states (see Fig. 4). This leads to a large number of control states, and to many reset transitions (for each intermediate state). For example, the transi-

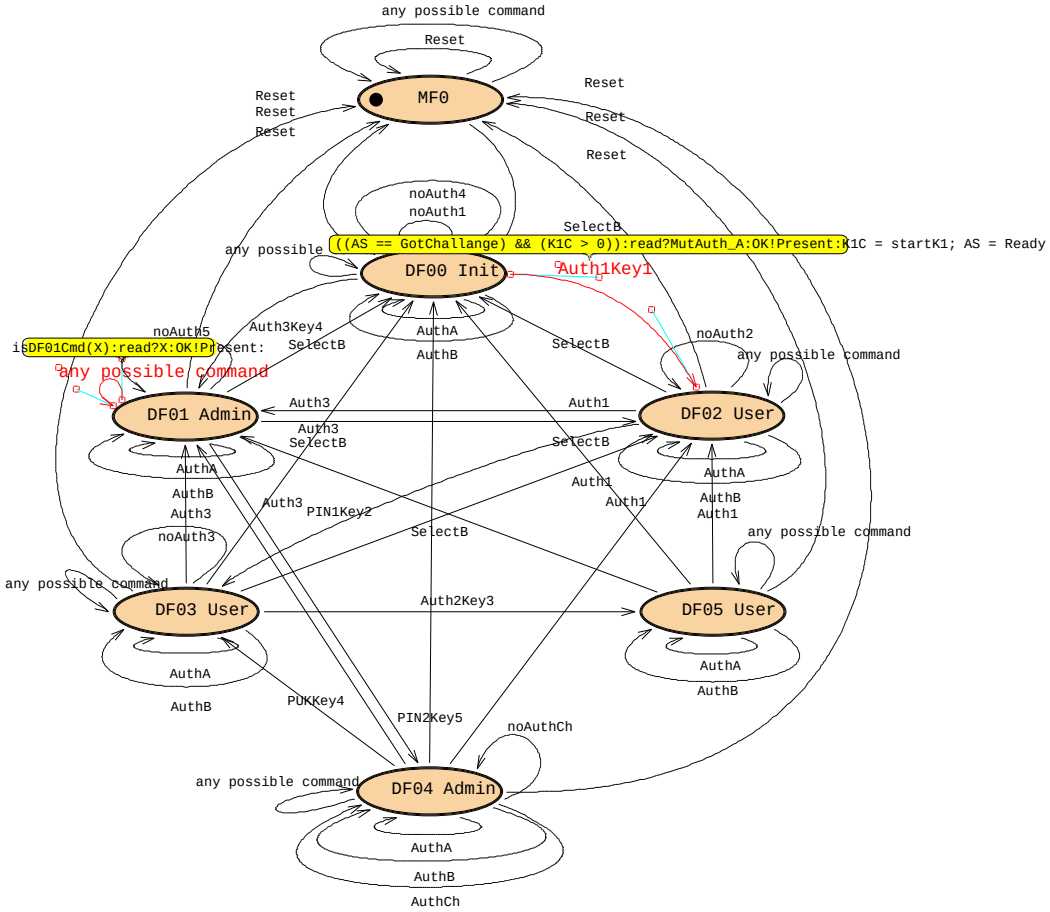

Figure 3: Inhouse Card: authentication with data states

tion that the card model fires between the intermediate state `I2` and the application state `DF02User` is `K1C>0; read?MutAuth_A; OK!Present; K1C=startK1`.

The application view encodes the intermediate states into a variable `AS:AuthState` of the type `data AuthState = Ready | GotData | GotChallenge` (Fig. 3). This reduces the number of states and reset transitions to the application states, and the STD is now very similar to the informal diagram in the given requirement document. In this case, the transition that takes care of the identification of the terminal is `AS == GotChallenge && K1C>0; read?MutAuth_A; OK!Present; AS=Ready; K1C=startK1`.

The last modeling task is to differentiate between the possible and the impossible commands at the different states. This is done via the "any possible command" transitions (see Figure 3): `isDF01Cmd(X); read?X; OK!Present`. This transition acknowledges all commands that satisfy the predicate `isDF01Cmd` with the value `Present`. The predicate `isDF01Cmd` describes the com-

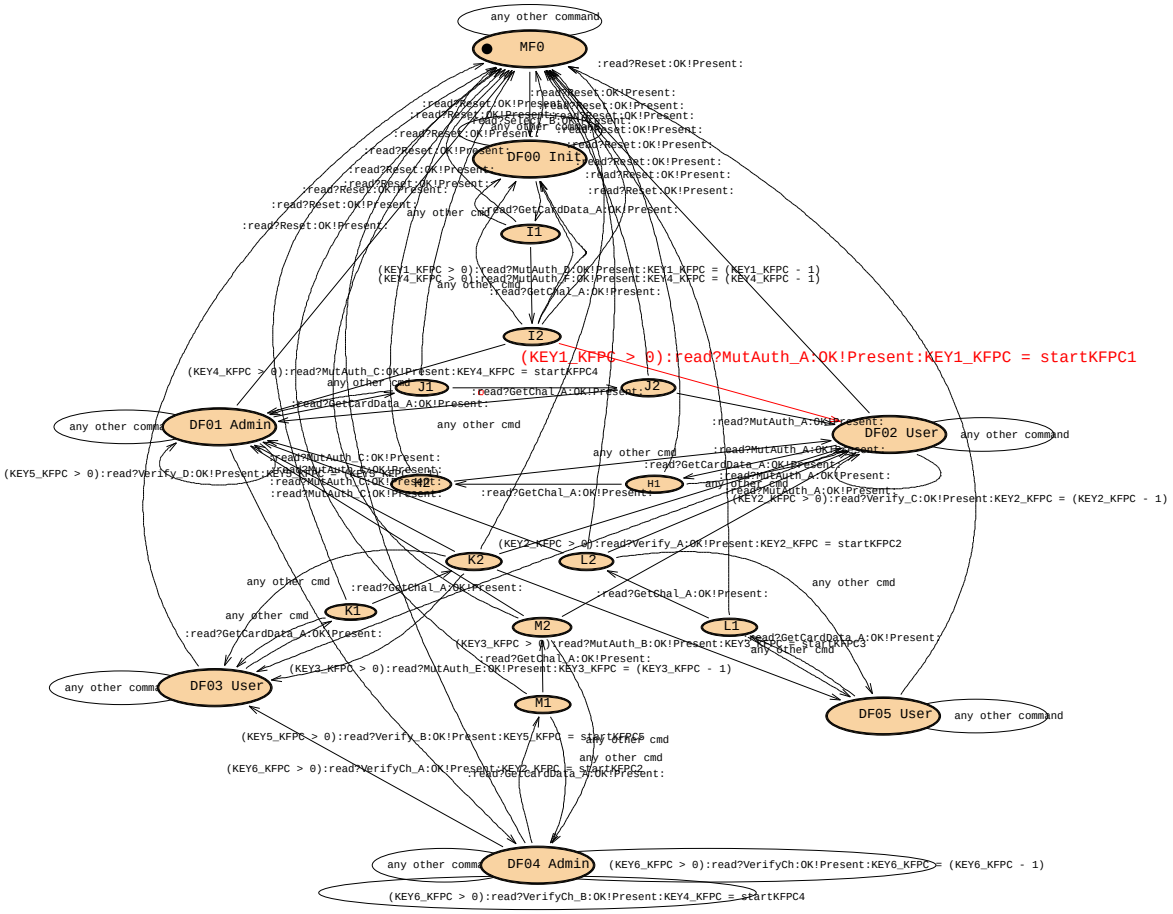

Figure 4: Inhouse Card: authentication with control states

mands that are admissible within the authentication state `DF01Admin`. The predicate is defined by

```
fun isDF01Cmd(ReadBin_A) = True
 |  isDF01Cmd(UpdateBin_E) = True
 |  ...
 |  isDF01Cmd(X) = False; // no others
```

This denotes that in these states successful reading is allowed (described by the equivalence class `ReadBin_A`), whereas updating results in an error (command equivalence class `UpdateBin_E`). For every state such a predicate is defined. This allows a very flexible modeling process since every command can be allowed or excluded explicitly without changing the layout and the transitions of the STD.

4 Model Based Testing

In this section, we describe our approach to model based testing on the grounds of Constraint Logic Programming. We restrict ourselves to a coarse description of the basic ideas; details of the translation may be found in [21, 22]; the embedding in an incremental development process is discussed in [28]. [27] contains a detailed discussion of our approach. While for the presented smart card example

the exact nature of the development process is not crucial, we think our ideas on test case generation are clarified by their embedding in the process.

4.1 Process

Many modern SW development processes emphasize the benefits of an iterative, or incremental, proceeding. These include, for instance, the Rational Unified Process (RUP [20]), Extreme Programming (XP [2]), the Cleanroom Reference Model (CRM [29]), or more classical prototyping approaches. Their main benefit is usually seen in the possibility of early interactions with the customer.

Similar to white box level specifications (i.e., state machines) of the CRM, we advocate the use of high level graphical specification languages as implemented in AutoFocus. The RUP as well as Extreme Modeling (XM [3]) focus on *models* as the integral entity of specifications. A model is an artifact that abstracts reality by focusing on specific aspects of it.

While the above processes necessitate a manual step from specifications to implementations, the code generators in AutoFocus permit (partial) automatization of this step. This is clearly due to the focus on *behavior* models–which are also in integral part in the UML-RT. However, we prefer the formally defined semantics of AutoFocus that we consider crucial in being able to compute test cases. This also is the difference to XM which, due to using statecharts in the UML, focuses on modeling rather than validation. In the sequel, when referring to models, we hence speak about behavior models.

In an incremental development process, test generation techniques naturally lend themselves to their application in regression testing [28]. With suitable management tools, test cases that have been derived for an earlier increment may be used for regression testing later ones. Note that we do not give a suitable definition of "increment" here that extends beyond "additional functionality"; this definition and its embedding in the development process is the subject of ongoing work.

Furthermore, when code generators do not satisfy requirements that are posed upon production code, test cases for models can, after suitable transformations, be used as test cases for the respective hand-written code. This naturally leads to the question of how test cases on models (functional, as in XP, or structural, i.e., satisfying some coverage criterion) relate to test cases on implementation code. This question is particularly important for structural test cases when certification issues enter the game. Our vision is to use structural test cases for models and to transform them into test cases on implementations by maintaining the respective coverage criterion (or switching from a suitable criterion on models to a suitable one on implementations).

We thus see the application domain of our approach in both testing models (as a debugging aid for error location) as well as testing implementations (where the specification is used as an oracle), and we do recognize the need for structural tests of implementations. We do not, however, oppose the view of F. Brooks [5]: "I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation."

4.2 Test case generation

The basic idea behind our algorithm is a symbolic execution of the model (see [21, 22] for details). To this end, we generate a set of predicates, $P_{\mathcal{A}}$, for each automaton, $\mathcal{A}$. Remember that automata only occur in bottom level components. Each predicate in $P_{\mathcal{A}}$ encodes one transition, and its arguments con-

tain thus the state and the destination state. Furthermore, the predicates' arguments contain formal parameters for input and output as well as for local variables. Guards and postconditions are encoded in the predicate's body, and they are evaluated to see if the encoded transition may fire or not, and how local variables are updated. The predicates in $P_{\mathcal{A}}$ are hence of the form

```
step_A(Src,Tr,Loc,In,Out,Dst):-
  pre(Tr,Loc,In),
  post(Tr,Loc,In,Out).
```

where `pre` and `post` encode guards and postconditions (assignments) for transition `Tr` from state `Src` to state `Dst`, taking into account values of local variables `Loc`, input values `In`, and output values `Out`. Both pre- and postconditions may involve arbitrary constraints. In practice, `Loc`, `In`, and `Out` are tuples of pairs rather than tuples of values. These pairs encode the values of the variable before and after firing transition `Tr`.

Composition. Now when composing a set of components, $\mathcal{C}$, by putting them in a new SSD and connecting their ports, a driver predicate d is needed. This predicate subsequently calls the predicates that correspond to the state machine of each of the elements of $\mathcal{C}$. Furthermore, it takes care of the communication between two components. Since internal channels—channels that connect two components—can be encoded by internal variables, the interface of the driver predicate d is, in terms of its structure, exactly the same as that of any bottom level component. Rather than storing pairs of values before and after firing a particular transition (or rather a set of them, since components fire simultaneously), we store the complete histories; this is done since we are interested in complete traces that are used as test cases. Histories are lists that contain the values of the particular channel or variable at each single tick. In this way, it is possible to update local variables or output channels with a value by simply concatenating this very value to the respective history list (remember that, when containing free variables, parameters in Prolog are always transient, i.e., the "free" parts are passed by reference). The predicate's head thus contains a tuple of lists for input histories, a tuple of lists for local variables, and a tuple of lists for output histories. The components of the tuple correspond to involved system's components; and since there may be more than one input/output channel or local variable, these again usually are tuples.

In this way, composition for more than two levels of hierarchy is achieved by applying this composition procedure recursively. For the top level component, it is necessary to take care of stimuli from the environment (input channels). Channels can only connect ports within one SSD: On a modeling level, inter-SSD communication consists thus of many chunks that connect a port of one atomic SSD with the interfaces of a set of SSDs, and penultimately, with an interface point of the SSD that contains an atomic SSD. The connection to this atomic SSD then forms the final chunk. Joining all chunks yields a "conceptual" channel that connects two atomic components. Communication over this channel is achieved in one tick.

Note that this simple translation scheme only works because of the simple synchronous communication semantics of AutoFocus. However, when necessary, we model asynchronous communication by explicit buffer components.

Execution. Symbolic execution now means successively calling the top level driver predicate. It usually is a good idea to restrict the

maximum possible length of the system runs. The exact number can be crucial in terms of efficiency [28], and its determination is an important task (which we do not consider further here).

Stimuli that are known (e.g., if they form part of a test case specification) can be inserted in the input history list; the same holds true for the values of output channels or internal variables. The backtracking mechanism ensures that if potentially, more than one transition of a particular automaton can fire, all of them are tried. This also ensures that if a predetermined output cannot be achieved by choosing a particular transition, all other possible transitions are tried. In addition, the use of free (i.e., unspecified or unbound) variables in Prolog enables one to compute test cases: Unspecified stimuli in an execution are encoded by those free variables. The choice of a transition by the respective driver predicate, scheduled by Prolog's backtracking mechanism[2], then binds this variable to a concrete value. In this way, completely instantiated system traces are computed. Thus far, what happens is an explicit generation of the system's state space, including the traces that led to each state.

However, this is too simple to work. The problem with Prolog's depth first strategy is that, whenever possible, it executes transition predicates in the same order as they have been written down. This results in loops - when two transitions emanate from a control state, the first of which leads to this very state, the second transition will only be taken when backtracking is performed. However, the problem becomes obvious if traces of a length of 10,000 or more are taken into account. We implemented two solutions to this problem. One consists of simply memorizing for each state which transition was last taken out of it, and when the state is re-entered, another transition is chosen. Our second implementation is based on probabilities for transitions that influence the choice of which transition is tried out first. As with probabilistic models in the CRM, the source of the transition probabilities is usually rather esoteric. However, in cases like the one mentioned below, it is a good idea to try a 50-50 probability without knowing what happens in the real system.

Constraints. Consider the guard of a transition that merely requires a local variable to be inside a certain range at a given point in time, t, for instance, $v_t > 3.2$. If the local variable v_t was hitherto unbound, it could, in principle, be bound to a value such as 3.2001. However, this instantiation is not necessarily essential: the system may well continue its execution with the knowledge that $v_t > 3.2$. This kind of information that accompanies the computation is called a *constraint*. If later on, for a particular trace, it turns out that v_t should indeed have been greater than, say 5.0, the corresponding constraint is updated to $v_t > 5.0$. However, it is also possible that later on, it turns out that the particular trace becomes impossible with $v_t > 3.2$, and that rather $v_t = 0$ would have been necessary. In such a situation the computation is discarded: the particular constraint is not satisfiable.

The bad news in this case are that obviously something went wrong, and backtracking is to be performed. The good news are that we do not need to continue the computation for we know that the constraint on v_t cannot be satisfied. This results in an a-priori pruning of the search tree, as opposed to the usual generate-and-test strategy that is so common in Logic programming.

In order to do so, some additional steps have to be taken. Since AUTOFOCUS allows for the

[2]Choosing a good ordering for the transitions gives rise to different search strategies like best-first on the grounds of appropriate fitness functions [26].

definition of (recursive) data types and functions that are used in guards and postconditions, we have to translate all data type declarations and function definitions into some kind of constraint (incidentally, $>$ is a predefined constraint in most CLP systems). This is done by means of Constraint Handling Rules—CHR [15]—, a meta language for constraint handlers (constraint handlers are those parts of the system that take care of checking satisfiability of constraints). Function definitions and calls are compiled into flat (eager) constrained predicates; since the generation of test cases does involve function inversions, we introduced an upper bound for the number of recursions in order to avoid infinite loops. Lazy evaluation is achieved by means of (delaying) constraints.

Note that constraints are not only used for representing sets of I/O but also for space efficient storage of sets of states.

Constraint instantiation. The last piece of the puzzle is concerned with an answer to the question of what to do with remaining constraints. The point is that a constraint such as $i_t > 3.2$ for an input value i_t for a given tick t may lead to an execution trace that satisfies the given test case specification (for instance, a particular coverage criterion). The test case specification can thus be satisfied without further restricting the value of i_t. In other words, we have just computed not one test sequence but a whole set of them: all those traces where input i at time t is greater than 3.2. This kind of situation naturally occurs with other constraints for (user defined) types other than the real numbers. The question than is to choose one value for i_t out of the infinitely many possibilities for a significant test case. In this kind of situation, heuristics have been developed (in this case, so called equivalence class heuristics: one would try three values, $3.1, 3.2$, and, say, 5.0). Naturally, clever instantiations are the major problem in the generation of test cases. Note that this approach differs from that in, for instance, [31] in that we deliberately use heuristics for the determination of test cases and not a theoretical approach that is based on the definition of a hopefully suitable notion of observation.

4.3 Discussion

The difference between the presented approach to test case generation and (possibly bounded, non-symbolic, on-the-fly) model checking becomes apparent in the handling of infinite systems. The relationship between (C)LP and model checking of (infinite) systems has been the subject of recent work [9, 12, 8, 6]. While the intention between testing and model checking is different in terms of intended completeness of the result, we see one major advantage of our approach in the higher flexibility in generating counter examples.

The use of constraint languages is also crucial in the intended interactivity which we consider the key to scalability of our method [27]: abstracting the model by excluding certain system runs, transitions, or states, is achieved most easily. In terms of performance, the predetermined maximum length of the generated test cases plays an important role: [28] contains an example where changing the maximum length by as little as twenty results in a change of as many as four orders of magnitude in terms of time needed to compute the specified test case. Since the reason for this behavior is Prolog's search strategy, better performing strategies for an intelligent choice of transitions (or, more generally, better search strategies than depth-first search with simple prevention of loops or probabilistic approaches) are thus needed. Our current work aims at such strategies; one of the ideas consists of using the topological structure of the automata for determining fitness functions for search strategies

as implemented, for instance, in the A* algorithm. In [26], we show how a fitness function for a specific class of test cases, namely reaching states or transitions, is defined by means of a reordering of the order in which transitions are chosen.

A yet unanswered question is that of appropriate input languages for test case specifications. Constraint languages (e.g., CHR) as an input language certainly are not always the best choice. Graphical input languages, such as sequence diagrams or automata, are probably better suited for a certain class of test cases. However, constraint languages like CHR seem to be a good choice as back end of such interfaces.

By now, the instantiation of remaining constraints is done on the grounds of simple heuristics or random instantiations. The question of how the created sequences relate to the model to be tested lies at the heart of our theoretical investigations. This is particularly interesting for impossible traces that occur, for instance, in robustness tests or in the application of test cases that have been computed for single components, to a composed component.

4.4 Application

We generated several test cases for this example. The objective was to derive test sequences for several coverage criteria (states, transitions) as well as for functional purposes. We restrict ourselves to giving numbers for the (smaller) version of the smart card where the authentication protocol is implemented by data states (local variables) rather than by control states. Memory requirements for the generation of all test cases was smaller than 10 MB; we omit the details. All measurements have been performed on a SUN UltraSPARC, 1GB of memory, 400MHz.

One of the test purposes was to achieve state coverage. Our system computed the corresponding test cases in $< .01$ seconds, for a given maximum length ranging from 10 to 100. The constraint specifying this test case is a macro `cover_states` that (automatically) rewrites to a set of membership constraints on the history of states that are being visited during execution.

For the sake of random testing, we made the system compute a test sequence of length 1000 (which does make sense with the above mentioned interleaving of transitions). The first sequence took 11.3 seconds to compute; subsequent ones could be obtained immediately.

Finally, Fig. 1 shows some experimental data for functional tests that required the counters to reach zero. For counters 1 and 2, the system could quickly determine the required test sequences. For counters 3, 4, and 6, this was not the case; we stopped computations after 2 hours. The reason for this behavior was easily found: the number of transitions emanating from each state is, with the strategy of interleaving transitions, too large. For counters 3 and 4, for instance, it is necessary that between two decrements exactly the same (looping) transitions have to be taken. With interleaving, it is not exactly a surprise we did not find the test case.

cnt #	from-to	max steps	time [s]
2	3-0	11	11
3	14-12	15	59
3	14-0*	10	.66
4	14-12	15	172
4	14-0*	43	.46
6	15-14	8	118
6	15-0*	20	.16

Table 1: Required time

In a first step, we thus made the system decrement the respective counter by 1 (or even 2), a task the system could handle. Since

we knew that in order to compute a trace where the counter reaches zero, it is sufficient to remain in the same state (because of the above mentioned looping transitions), we simply specified the respective test case as follows: First, make the system decrement the respective counter by one, and then, remain in the same state until the counter reaches zero. We thus "sliced" the model by ad-hoc restricting ourselves to one particular state (with all looping transitions enabled). In this way, it was again simple to finally compute the test sequences; the lines in the table with a $*$ indicate that we helped the system in the described manner.

It is noteworthy that with the other selection strategy of choosing transitions by means of given probabilities, we were able to find all test cases without "helping" the system (in reasonable times, below five minutes). In a way, however, this is a workaround: If the loop transitions in questions are given exorbitantly high probabilities *because we know what the problem is*, what we actually do is not different from forbidding certain states or transitions. This shows, however, that with knowledge of the system, it is possible to even compute "difficult" systems, and, again, we consider the possibility of interaction as the key factor in scalability of our approach as well as in its graceful degradation [27]. In [26], we show the fully (and almost instantaneous) automatic determination of the test cases in Tab. 1 with a combination of best-first and tabu search where, as mentioned above, the fitness function is defined by means of shortest paths in the state machine and is implicitly implemented by a transition reordering. Tabu search is implemented by different strategies for storing sets of already visited states.

It might be interesting to note that in the case of counter 6, we were able to find the corresponding test case without hints with one of the model checkers connected to AutoFocus, SMV (i.e., a trace t with $t \models \Diamond \texttt{Card.K6C} = 0$). For a discussion of the relationship of (bounded) model checking and testing, we refer to [27, 26]; consider also the remarks in the paragraph on related work in the first section. For the model checking approach we are quite close to the complexity limit (state explosion problem). The application (modeler's) model has 38 (38) state-bits with 54 (72) transitions. Model checking with SMV required 30 (20) seconds, 112464 (73145) BDD nodes and 3 (2.5) MB storage. Bounded model checking [32] with SATO fails to find these examples. Our approach with best-first search (see above, [26]) succeeds in finding the test case in less than 0.01 seconds.

5 Conclusion

We have presented some results of a feasibility study that aimed at assessing the practicability of a test case generator on the grounds of Constraint Logic Programming. The results show that the test case generator in combination with a suitable modeling tool like AutoFocus allows to compute relevant test cases for industrial applications. This alleviates the tedious task of test developers. In fact, the authors' institutions agreed to continue their cooperation. In the remainder we briefly assess both the modeling capabilities of AutoFocus and the test generator.

Modeling. Specification formalisms, GUI, and tool support of AutoFocus were perceived to be easier to grasp and more comprehensive than other approaches used in previous studies, e.g., product nets. The possibility to quickly alter a model and to be able to immediately (i.e., after compilation) simulate it was also considered to be very helpful. The possibility to "replay" or simulate computed test cases interactively is most important for indus-

trial testing. In addition, the integration of the modeling and the testing tools was identified to be crucial.

Test case generation. In addition to actually generating complete test sequences (from specifications such as "transition tour"), it is important to verify that a given test sequence satisfies the intended test purpose (formalized by a test case specification). The approach presented in this paper obviously facilitates this task - computed test sequences do what they ought to by construction, i.e., conform to their test case specification.

The ability to formulate arbitrary test case specifications by means of Constraint Handling Rules is considered to be one of the strengths of this approach. However, this requires expert's knowledge, and the tradeoff between the tool's computation power and interaction is acknowledged. Nonetheless, formulating test case specifications by means of CHRs is considered to be rather acceptable.

In terms of current research, we are focusing on the definition of fitness functions for A*/best first search strategies for different classes of test cases. This also includes reckoning procedures for values of the maximum depth of the search tree. Storing states by means of constraints directly lends itself to abstractions by considering convex hulls of sets of states rather than the exact sets.

Finally, the quantitative results in this paper clearly lack comparative numbers. This is in part due to the fact that the example is not a publicly available academic example (which does not, of course, mean that the system could not be re-modeled in Lustre, and that existing test tools such as Lutess [11] or Gatel [23] could not be used for test case generation). Another reason is that there are hardly any tools that can be used for graphical specification as well as for test case generation, a situation that, with the enormous industrial interest in such tools, is most likely to change.

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, February 1995.

[2] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 1999.

[3] M. Boger, T. Baier, F. Wienberg, and W. Lamersdorf. Extreme modeling. In *Proc. Extreme Programming and Flexible Processes in SW Engineering (XP'00)*, 2000.

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

[5] F. Brooks. No Silver Bullet. In *Proc. 10th IFIP World Computing Conference*, pages 1069–1076, 1986.

[6] T. Bultan. *Automated symbolic analysis of reactive systems*. PhD thesis, University of Maryland, 1998.

[7] 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.

[8] B. Cui, Y. Dong, X. Du, K. Narayan Kumar, C. Ramakrishnan, I. Ramakrishnan, A. Roychoudhury, S. Smolka, and D. Warren. Logic programming and model checking. In *Proc. PLILP/ALP*, Springer LNCS 1490, pages 1–20, 1998.

[9] G. Delzanno and A. Podelski. Model Checking in CLP. In *Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, pages 223–239, 1999.

[10] R. DeMillo and A. Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[11] L. du Bousquet and N. Zuanon. An overview of lutess, a specification-based tool for testing synchronous software. In *Proc. 14th IEEE Intl. Conf. on Automated SW Engineering*, October 1999.

[12] L. Fribourg. Constraint logic programming applied to model checking. In *Proc. 9th Int. Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'99)*, LNCS 1817, Venice, 1999. Springer Verlag.

[13] L. Fribourg and M. Veloso-Peixoto. Automates Concurrents à Contraintes. *Technique et Science Informatiques*, 13(6):837–866, 1994.

[14] T. Frühwirth. Constraint Handling Rules. In *Constraint Programming: Basics and Trends (LNCS 910)*, pages 90–107. Springer Verlag, 1995.

[15] T. Frühwirth. Theory and practice of constraint handling rules. *J. Logic Programming*, 37(1-3):95–138, October 1998.

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

[17] G. Gupta and E. Pontelli. A Constraint-based Approach to Specification and Verification of Real-time Systems. In *Proc. IEEE Real-time Symposium*, pages 230–239, San Francisco, December 1997.

[18] F. Huber, B. Schätz, and G. Einert. Consistent Graphical Specification of Distributed Systems. In *Industrial Applications and Strengthened Foundations of Formal Methods (FME'97)*, LNCS 1313, pages 122–141. Springer Verlag, 1997.

[19] International Organization for Standardization. International Standard ISO/IEC 7816: Integrated circuit(s) cards with contacts, 1995.

[20] P. Kruchten. *The Rational Unified Process: An Introduction*. Addison Wesley, 2nd edition, 2000.

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

[22] 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.

[23] B. Marre and A. Arnould. Test Sequence Generation from Lustre Descriptions: GATEL. In *Proc. 15th IEEE Intl. Conf on Automated Software Engineering (ASE'00)*, Grenoble, 2000.

[24] C. Meudec. ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution. In *Proc. 1st Intl. workshop on Automated Program Analysis, Testing, and Verification*, Limerick, 2000.

[25] O. Müller and T. Stauner. Modelling and verification using Linear Hybrid Automata. *Mathematical Computer Modeling of Dynamical Systems*, 6(1), March 2000.

[26] A. Pretschner. Classical search strategies for test case generation with Constraint Logic Programming, 2001. Submitted to FATES'01.

[27] A. Pretschner and H. Lötzbeyer. Model Based Testing with Constraint Logic Programming: First Results and Challenges, 2001. Submitted to 2nd ICSE Intl. Workshop on Automated Program Analysis, Testing, and Verification (WAPATV'01).

[28] A. Pretschner, H. Lötzbeyer, and J. Philipps. Model Based Testing in Evolutionary Software Development. In *Proc. 11th IEEE Intl. Workshop on Rapid System Prototyping (RSP'01)*, Monterey, June 2001. To appear.

[29] S. Prowell, C. Trammell, R. Linger, and J. Poore. *Cleanroom Software Engineering*. Addison Wesley, 1999.

[30] V. Rusu, L. du Bousquet, and T. Jéron. An Approach to Symbolic Test Generation. In *Proc. Integrated Formal Methods*, 2000.

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

[32] G. Wimmel, H. Lötzbeyer, A. Pretschner, and O. Slotosch. Specification Based Test Sequence Generation with Propositional Logic. *J. Software Testing, Validation, and Reliability*, 10(4):229–248, 2000.