Int J Softw Tools Technol Transfer (2004) / Digital Object Identifier (DOI) 10.1007/s10009-003-0128-3

Model-based testing for real

The inhouse card case study

A. Pretschner[1,*], **O. Slotosch**[2], **E. Aiglstorfer**[3], **S. Kriebel**[4]

[1] Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland
e-mail: pretscha@inf.ethz.ch
[2] Validas Model Validation AG, gate, Lichtenbergstr. 8, 85748 Garching, Germany
[3] Giesecke&Devrient GmbH, Prinzregentenstr. 159, 81667 Munich, Germany
[4] BMW Group, 80788 Munich, Germany

Published online: 2 March 2004 –

Abstract. Model-based testing relies on abstract behavior models for test case generation. These models are abstractions, i.e., simplifications. For deterministic reactive systems, test cases are sequences of input and expected output. To bridge the different levels of abstraction, input must be concretized before being applied to the system under test. The system's output must then be abstracted before being compared to the output of the model.

The concepts are discussed along the lines of a feasibility study, an inhouse smart card case study. We describe the modeling concepts of the CASE tool AutoFocus and an approach to model-based test case generation that is based on symbolic execution with Constraint Logic Programming.

Different search strategies and algorithms for test case generation are discussed. Besides validating the model itself, generated test cases were used to verify the actual hardware with respect to these traces.

Keywords: Test case generation – Model checking – Symbolic execution – Behavior models

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 [31] with a prototype for the automatic generation of test cases on the grounds of symbolic execution with Constraint Logic Programming (CLP) [36]. 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 are to be assessed. The main result of this study is that for this application domain, the techniques in question – symbolic execution with state storage and directed search – are convincing candidates for further exploration, as expressed in subsequent projects of the authors' institutions. These projects target the generation of test cases for further chip card applications such as the wireless identity module used for card holder verification and cryptographic operations such as computing digital signatures; the results have been published elsewhere [43]. This paper is an extended version of a workshop contribution [51] augmented by additional material from earlier work [46].

Overview. The paper is organized as follows. Section 2 contains an introduction to our understanding of model-based testing. The role of different models at different stages of development is highlighted. In Sect. 3, we briefly present the CASE tool AutoFocus for specification, simulation, and validation of reactive systems. Section 4 then describes the case study. In Sect. 5, the core of this paper, we describe and discuss our approach to the generation of test cases and present some experimental results. An approach to solving the problem of storing sets of states in the context of symbolic execution is presented. The paper concludes with an assessment of the results of this feasibility study that takes into account both the modeling formalisms of AutoFocus and the generation of test cases.

Related work. In large parts, the following discussion is taken from earlier work [49].

The general schema of generating test cases is similar to Howden's proposition [28]. The intricate question of what should be tested [23, 65] remains unsolved. Zhu et al. review the use of structural coverage criteria as test case specifications in their overview article [67]. Ntafos compares several control flow coverage criteria [41], and

* The work described in this article was carried out while the first author worked at Technische Universität München.

Frankl and Weyuker do so for data flow testing criteria [20]. Vilkomir and Bowen provide a formalization of control flow coverage criteria that can serve for the automatic generation not of test cases but rather test case specifications [60]. Finally, random testing as test case specification turns out to be powerful whenever input data partitioning – and consequently structural coverage – is not based on a priori knowledge of the likelihood of errors in the different partitions [17, 25, 27].

The following discussion contains an overview of the state of the art in test case generation techniques for reactive systems. Test patterns and test case generation for object oriented systems are deliberately not reviewed [2].

Synchronous languages. Much of the literature on test case generation for reactive systems is related to the French school of synchronous programming languages. This is no coincidence: their precise semantics and operational model lend themselves well to algorithmic analysis.

The Lurette tool [54] aims at generating black-box test sequences for Lustre programs. The idea is to encode the notion of "relevant" behaviors (or test sequences) in an observer node – basically, a set of constraints over Booleans and real variables that take into account the directly preceding outputs of the program under test. These are used for the computation of new inputs to the program: random values that satisfy all constraints.

The Lutess tool [15] follows the same basic ideas for black-box testing invariance properties. I/O data types are restricted to the Booleans; the "observer" (or test case specification, respectively) is encoded as a BDD. At each step, a value satisfying the constraints given by this BDD is chosen. Lutess provides several strategies for property-based, random, stochastic (environment) profile-based, and behavioral pattern-based choice of inputs.

The GATeL tool [38] computes test sequences for Lustre programs by solving systems of constraints. These constraint systems directly reflect the Lustre dataflow equations; heuristics are used for choosing which variables to instantiate and when. Constraint instantiation may proceed at random; backtracking is provided by the underlying CLP system. Structured data types as well as recursive functions on the "transition" level are not supported.

LOTOS/SDL/Z/(E)FSMs. The TGV tool [19] computes test sequences for LOTOS and SDL specifications translated into Input/Output Labeled Transition Systems. Essentially, the synchronous product of a test purpose and the system under test is traversed in order to decorate this product with verdicts and timers. The result is a test graph (representing test sequences and their respective verdicts). Data states in the IOLTS are not explicitly taken into account.

Like TGV, some of the test case generators connected to TorX [14] build on the so-called ioco testing theory [58] for labeled transition systems. TorX is a test tool architecture that allows black-box batchwise and on-the-fly testing of SDL, LOTOS, and Promela specifications. The generation algorithm for on-the-fly testing is similar to that of Lurette or Lutess in that they are generated along with corresponding verdicts while traversing a product of system (I/O) and test case. Rusu et al. explicitly extend IOLTs with variables [55]. The basic ideas of the underlying test case generator are similar to that of TGV (product of test case specification and system under test). The motivation for this is not to explicitly encode a system's data state in an LTS that severely suffers from the state space explosion problem.

The approaches based on IOLTS as an intermediate language require the explicit computation of the specifications' product automata (specifications tend to consist of more than one process or, in the presented context, component). In addition, it necessitates computation of this product automaton with the test case specification. In many cases, this leads to a large number of unsatisfiable transition guards that, during state space exploration, have to be computed for each product transition. Rusu et al. use PVS and HyTech for the expensive automatic computation of invariants that may enable one to statically exclude these transitions [55]. Current work concentrates on also including such static analysis procedures; note, however, that in the presented compositional approach, the above-mentioned failing guards need to be computed only once.

Peleska and Siegel explain how to use CSP models to perform real-time testing [42]. Tool support is available; there is a focus on environment models that are used for test case generation. The basic idea of using Hennessy's testing theory for real-time testing is also used by Nielsen [40]. AutoLink [34] is a tool for generation of test cases from SDL specifications. It works on LTS rather than extended finite state machines. No symbolic execution for reducing the state space is performed.

Bourhfir et al. present different test generation techniques for EFSM-based systems [3]. Chow [7] discusses comprehensive testing of finite state systems where large sets of test cases converge against proofs. Ural gives an overview of different testing strategies for finite state machines [59]; Bochmann and Petrenko discuss their use in the domain of protocol testing [62]. Among others, Sadeghipour and Burton [6, 56] use the Isabelle theorem prover for generating test cases from Z specifications with extended finite state machines or StateCharts.

Model-checking-based approaches. The obvious idea of directly using counterexamples from (symbolic as well as explicit-state) model checkers as test cases has been considered widely; see, for instance, the work of Ammann et al. or, for the use of bounded model checking, that of Wimmel et al. [1, 66].

While not directly aiming at testing, Edelkamp et al. augment SPIN's search strategies with directed search [18]. Fitness functions are based on the degree of satisfaction of formulas and on the euclidian

distance between states (in the context of evolutionary testing and with a slightly different motivation, this approach is also taken in the theses of Tracey and Wegener [57, 64]). The structure of the never-claim is taken into account for increasing efficiency when model checking liveness properties. Conceptually, this approach is quite similar to the presented efforts in using best-first heuristics; the difference lies in the use of constraints for symbolically executing several traces simultaneously and for storing large sets of states as well as in the methodology: it is not stated exactly which properties are to be checked (i.e., how to derive a set of finite test sequences for, e.g., an invariance property).

The Java PathFinder project [61] deploys explicit model checking technology for analyzing Java programs with similar heuristics [24]; due to the infinite state space of such programs, this is an approach similar to ours (differing in the technology as well as the emphasis on existential properties).

Constraint- and logic-based approaches, symbolic execution. The approach of Ciarlini and Frühwirth [8] is similar to ours (symbolic execution on the grounds of CLP), differing from the presented work with respect to compositionality, the degree of automation, and the combination of state machines with specifications in a functional language for guards and postconditions, i.e., the input language. State storage is not taken into account.

Fribourg notes the resemblance of constraint solving in Logic Programming and model checking of (infinite state) systems [21]. Delzanno and Podelski identify logical implications in LP with transitions and, with a set of sound widening abstractions, model check several infinite state systems in this way [12]. Cui et al. use tabled resolution [63] in XSB Prolog for the implementation of an efficient model checker [11].

Symbolic execution for test case generation has been discussed for a long time. Examples include the work of Clarke, Howden, King, and Ramamoorthy et al. [10, 29, 30, 33, 53]. Because performance constraint solvers were not readily available at the time, these systems often needed help in terms of, for instance, prespecified paths.

Meudec uses CLP for symbolic execution of SPARK (SPADE Ada Kernel) code to compute test cases [39]. The focus is neither on reactive nor on concurrent systems. In fact, SPARK is used for high integrity software and deliberately does not support tasks. Concurrency has to be implemented explicitly by writing appropriate schedulers.

Legeard and Peureux use a constraint solver to compute test cases from B specifications [35]. While similar to the presented approach, there are some technical differences. Constraints are not used for storing hitherto visited states. Directed search does not seem to be implemented. The authors' experience is that these techniques are necessary to get an acceptable performance, even for small systems. The application domain is, as in our case [43], that of chip cards.

Logic variables naturally lend themselves to be used in symbolic execution and test case generation. Consequently, there are many Prolog-based approaches. Denney, for instance, abstracts recursion in Prolog programs and heuristically defines equivalence classes on the resulting abstract flow graph that are then used as a basis for symbolic execution [13]. Since Prolog is not typed, all these approaches do not take into account types. Types, however, are taken into account in the QuickCheck tool for (functional) Haskell programs [9]. The basic idea is to give a separate specification of a program and randomly generate test cases. AutoFocus makes use of an eager functional language; testing functional programs is thus one part of the presented approach.

For the sake of simplicity, we synonymously speak of test cases and test sequences as sequences of I/O traces of a system. Terminological issues are discussed in Sect. 2.3.

2 Overview of model-based testing

Testing denotes a set of activities that aim at providing evidence that the behavior of an implementation does or does not conform to its intended behavior. The latter tends to be incompletely represented in usually informal specification documents. Knowledge of these documents enables test engineers to build a vague understanding of what the system is supposed to do; they build a mental model. It turns out that relying on these implicit models renders the process of testing unstructured, barely reproducible, and unmotivated in its details.

2.1 Model-based testing

The idea of model-based testing, then, is to use explicit behavior models instead. They encode the desired behavior and are used to derive test cases that are used for both

- validating the model, i.e., manually checking every single test case, which is necessary if a further formal specification does not exist, and
- verifying the respective implementation, which requires bridging the gap between different levels of abstraction. This is because models are simplifications, and they are meant to be simplifications in order to be able to intellectually master the artifact in question.

In this paper, we consider deterministic reactive systems.[1] For deterministic systems, test cases can be seen as sequences of input and output. The output of the model encodes the desired behavior of the implementation. After suitable concretization, the model's input is fed into the implementation, and, after suitable abstraction, the implementation's output is compared to the model's. This

[1] More precisely, we consider deterministic models that can, in some cases, be used to test even nondeterministic systems.

allows an automatic assessment of whether or not the implementation does what it is intended to do. Again, the intended behavior is assumed to be given as a validated model.

One may well ask why validating the model and subsequently generating test cases for the implementation should be more effective and/or efficient than validating the system directly. We see the reason in the use of abstract, or simplified, models. Simplified models are more likely to be intellectually manageable than entire systems. In addition, using models means, for instance, that automatic test case generation for regression testing becomes possible. On the other hand, missing information must be inserted at some point. This is achieved by means of driver components that take care of concretizing input and abstracting output. By doing so, complexity is distributed between the model and the driver. Clearly, choosing adequate levels of abstraction is a challenging task.

Then, rather than manually identifying single test cases, the idea of model-based test case generation is to use a selection criterion that describes a *set of test cases*, aiming at testing a given test purpose (below we give a more precise terminology). This selection criterion is what we refer to as test case specification. The idea is thus to work at a higher level of abstraction: specifying whole sets of test cases instead of specifying each single test case, one after another.

2.2 Models

Models for data and communication. In many areas of software and systems engineering, models enjoy an increasing popularity: in the domain of business information systems, *data models* represented by class or entity-relationship diagrams have successfully been used for years. The OMG's model-driven architecture promotes the use of platform-independent models in order to abstract from concrete *communication infrastructures*. The idea is to (a) be able to develop systems without taking into account detailed communication issues right from the beginning and (b) reduce coding efforts by relying on generated code or code skeletons.

Behavior models. The idea behind models of *embedded systems* is to take these ideas further and to use abstract descriptions for the behavior of a system. When compared to business information systems, embedded systems tend to exhibit a more complicated control flow and less complicated data structures. Description techniques for behavior include SDL, process algebras, statecharts, finite state machines, or Petri nets. The problem with behavior models is that people want them to be abstract and concrete at the same time. They should be abstract to become manageable, and they should be concrete to render automated code or test case generation feasible. While data models are static, behavior models obviously are dynamic, which substantially adds to their complexity.

While one may well question the existence of underlying abstractions, the use of models in engineering embedded systems is already comparatively widespread. In the domain of continuous and mixed discrete-continuous systems, description techniques and CASE support have matured to a point where the generation of not only simulation code but also production code has become a reality. This is impressively demonstrated by tools like Matlab Stateflow/Simulink with associated production code generators (Beacon or Targetlink) or ASCET-SD in the automotive and avionics domains.

System and environment. Naturally, there are two kinds of models: models of an implementation and models of the environment. For test case generation, both are important: a model of the implementation specifies the intended behavior. In itself, it can be used for the generation of test cases. If it is sufficiently precise, then it can, at least in principle, also be used for code generation. One could use the same model for both generating production code and test cases. The generated test cases, however, cannot be used for testing the functionality of the implementation. Instead, they may be used for testing code generators, legacy code, or assumptions about the environment.

Models of the environment, on the other hand, are used to enforce implicit assumptions in the model of the implementation. This is usually necessary for simulation. Since environment models encode a set of scenarios, they can be used for test case generation, too, since they usually restrict the number of possible runs of a system.

Model and code. In this paper, we consider the model of an implementation that has been built *after* the implementation [50]. This was done in order to assess the benefits and shortcomings of our test case generation procedure. In the domain of model-based testing, this kind of scenario, however, turns out to be a rather important one. This is the case if a new device is to be tested in conjunction with existing legacy systems – just consider building the network master of, say, a multimedia bus system in modern vehicles. In order to generate test cases for the network master, we need models of the connected devices (CD player, tuner, navigation systems) that faithfully represent the reactions of the actual systems. Models of these connected devices are also necessary to stub the actual devices if certain behaviors of the actual network master are to be tested. The point is that models of connected devices are environment models. In the case of the chip card, such a model would be necessary to generate test cases for the respective terminal: the model of the chip card would then be an environment model.

If models are built *before* the system under test, they serve the double purpose of specifications and basis for test case generation. One must be careful, however, not to generate both production code and test cases from

a model – the system would be tested against itself. In this case, code generators and assumptions on the environment can be tested, i.e., the embedding of a system into its context of hardware, operating system, and legacy code. However, models for code generation must be very precise, while models for test case generation (or as specifications) can be more abstract. Missing information is inserted by the above-mentioned driver components when the test is actually executed [43]. If models are executable, they combine the advantages of rapid system prototyping with a more abstract treatment of relevant issues. Different scenarios of model-based testing are discussed elsewhere [50].

We thus see the application domain of our approach in 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 structured tests of implementations. We do not, however, oppose the view of Brooks [4]: "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."

2.3 Terminology

A test case is a structure of finite input and expected output. For deterministic systems, the structure is a sequence; for nondeterministic systems, it is treelike. In the case of deterministic systems, a test case then is a sequence of input and output *of a model*. It represents the intended behavior. A test case can represent many traces of an implementation. This is because of jitter in the time and value domains. In the chip card example of this paper, for instance, timing behavior is not relevant at the level of abstraction we consider. Jitter in the value domain is ubiquitous when continuous systems are taken into account.

A test case specification is an intrinsic description of a set of test cases (a test suite). Test case specifications reflect a given informal test purpose. Test purposes include "requirement A", "invariance property I", or "coverage criterion C". Obviously, test purposes cannot directly be used for testing. Test case specifications render test purposes amenable to the automatic generation of test cases:

- "Requirement A" may be expressed as an incomplete sequence diagram, and a test case generator inserts missing signals. It may also be expressed as an entire state machine that encodes all the relevant scenarios of a protocol (an environment model). In this case, the test case generator must, at random, for instance, choose some of the traces of the model that is the result of composing the models of the environment and the implementation.
- An "invariance property I" is difficult to test for it describes infinitely many traces. A corresponding test case specification may restrict them to a fixed length, and it may, in addition, require that no state of the model be entered twice when all test cases are considered.
- In order to obtain a test suite that satisfies a certain coverage criterion, a set of test case specifications must be provided. These specifications ensure, for instance, that a test case generator will generate traces such that each single statement is executed at least once, or that each condition is evaluated in a certain manner. For state machines, for instance, coverage criteria are defined w.r.t. transitions or states. In practice, coverage criteria are applied to both models of the system under test and models of the environment. The latter encode sets of scenarios, and they reduce the state space (see above).
- In addition to functional and structural (coverage) criteria, test case specification may also be of a stochastic nature. Test cases can then be generated at random or with respect to a given probability distribution.

Test case specifications are thus bound to a certain test case generator. In this paper, we consider test case specifications that amount to finding certain states or signals in system runs. In general, functional and structural test purposes can be broken down into a set of such test case specifications.

To summarize: rather informally, test purposes describe what is to be tested. Test case specifications formalize this in a manner such that a test case generator is able to compute test cases. These test cases are traces of the model, and since the model is an abstraction, they may represent many traces of an implementation.

How good are the generated test cases? We only answer this question indirectly: they are as good as one wants them to be. They are as good as the test case specification is. If one considers a coverage criterion to characterize a "good" test suite, then one can generate test cases that satisfy the criterion. The test cases are then "good". The problem is that there is no generally accepted notion of a "good" test suite. Note that stating that a test suite is "good" if it identifies failures of a system is not satisfying either: for a perfect system, a "good" test suite would not exist. Stating that a "good" test suite should identify *potential* errors is a better approach, but how is this to be quantified?

Testing and test case generation is more complicated for nondeterministic systems. The reason is that the test case may have to "react" to nondeterministic reactions of the system under test. A test case then represents a whole set of traces of the model, and it may well be given as a state machine. We will not further discuss nondeterministic models of an implementation in this paper. Note, however, that it is sometimes possible to use deterministic models for testing nondeterministic systems by coping with nondeterminism at the level of the above-mentioned driver components.

3 AutoFocus

AutoFocus (`autofocus.in.tum.de`, [31]) 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 time-synchronously communicate via typed and directed *channels*. Each end of a channel is connected to a *port*. 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 nonhierarchic component that is depicted in Fig. 1 (the complexity of our example system lies in the behavior, not in the structure; more complex studies are described by the authors [43–45]). 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. Figure 3 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 (i) certain pattern matching conditions of the form *channel?pattern* on the input channels hold and (ii) their guard, a condition on input values and the local data state, holds. Guards are expressed by means of possibly recursive functional programs. Firing then means to update local variables and write outputs. This is achieved by means of so-called assignments.

Note that in Fig. 3 transitions are simply labeled since the respective input statements, guards, and assignments would clutter the diagram. An example of a transition (from Fig. 3) is labeled with `any possible command`. It has the following annotations:

```
isDF01Cmd(X);read?X;OK!Present;
```

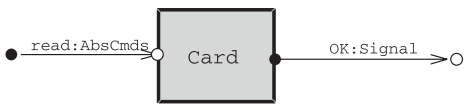

Fig. 1. Interface of inhouse card

This transition executes if a value is present on input port `read`, and the value (bound to the variable `X`) satisfies the guard predicate `isDF01Cmd`. In this case, the transition fires and the value `Present` is written to the output port `OK`. In this case, no local variables are updated.

Data and functions. Channels are typed. Allowed 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 is used for the definition of new, possibly recursive, functions (using the keyword `fun`). These functions are used in the guard or postcondition of a transition. Examples of user-defined data types and functions are given in Sect. 4.3.

Execution. AutoFocus components execute concurrently and simultaneously in a time-synchronous manner. The existence of a global clock ensures the existence of so-called *ticks*. Before each tick, every component reads its input ports. It then computes pattern matching conditions and guards for all possible transitions. Possibly nondeterministically, one of them is then chosen; if there is none, the system idles, i.e., remains in its current (control and data) state. Recall that for test case generation, in this paper we are only concerned with deterministic models. In addition, the model computes new values for local variables and output ports. During the (instantaneous) tick, it updates the respective variables 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.

Validation and verification. 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, NuSMV, μcke), propositional solvers for test case generation and bounded model checking (SATO, chaff) [66], and theorem provers (VSE, Isabelle) have been connected to the tool. In addition, tool integration with Rational UnitTest and DOORS for requirements tracing exists. Many Matlab-Simulink/Stateflow models can also be imported into AutoFocus.

4 The inhouse application

In this section we describe the inhouse application as an example of the integration of a smart card as a security device within an entire security framework. The inhouse application is used industrially for demonstration only.

However, in terms of its technical features, it can be regarded as a real-life application without any restrictions. The inhouse card conforms to ISO 7816 [32], which provides hardware characteristics such as size of the card and power supply, as well as the programmable interface for the application, and the card's life cycle.

4.1 Application

The inhouse card is a secure device within a security framework. The application can be arbitrarily designed to allow authorized access only, e.g., to a site, a building, parts of a building, rooms, or terminal access of a computer network. Each request for access is represented by a virtual security state that is achieved on the smart card by selecting a certain file of the chip card's file system. Roughly speaking, such a file system consists of directories (dedicated files, DFs) and elementary files (EFs). File access is allowed after valid authentication of the card reader terminal and/or verification of the card holder. Card holder verification is achieved by a personal identification number (PIN). The inhouse application described in the following is intentionally kept as simple as possible. Nonetheless, it also comprises the features also necessary for more sophisticated security concepts.

The file system is the core of a smart card application [32]. It provides all necessary information such as card holder keys and secret keys. The file system of the inhouse application used in the remainder of this paper comprises five EFs (EF00–EF04) and the internal secret file (ISF). The EFs are used for storage of application relevant data, the ISF provides all data for the necessary keys (keys 1–6). Each of the states in Fig. 3 corresponds to a certain set of privileges or admissions the user is granted once he has correctly authenticated himself.

The initial state MF00 of the smart card is entered by selecting the root directory of the file system, after each card reset or power off. As multiple applications can be stored on a smart card, the respective application is to be selected in this state. After selection of the file systems subtree where the inhouse card application is stored, state DF00 is reached. State DF00 allows read access for two of the data files (EF00 and EF04). The inhouse application now checks the desired access rights of the card reader terminal and the smart card by mutual authentication based on symmetric cryptography. Either user mode or administrator mode is provided by the inhouse application. All secret keys needed for the transition from one security state to another are secured by a so-called error counter. The keys for the mutual authentication procedures allow 15 consecutive key errors, those for the PIN procedures only 3. A reduced key error counter is reset when the respective authentication is successful. An error counter that allows no more consecutive errors locks the transition to the subsequent security state.

If the terminal requests user mode access, state DF02 is reached by mutual authentication without any card holder activity through key 1 and symmetric cryptography (DES). DF02 allows read access for the data files EF00, EF01, and EF04. The authentication procedure is then followed by card holder verification through key 2, the user PIN. The PIN transmitted through the card reader terminal is compared by the smart card with the key 2 stored on the card. If these match, security state DF03 is entered. Security state DF03 allows read access to all data files except for EF03, as well as write access to EF01. A further successful mutual authentication with key 3 triggers the transition from DF03 to DF05, which allows read access to all data files as well as write access to EF01.

If the terminal requests administrator mode, access state DF01 is reached by mutual authentication in state MF00 without any card holder activity through key 1 and symmetric cryptography (DES). DF01 allows read access to the data files EF00 and EF04. The authentication procedure is then followed by card holder verification through key 5, the administrator PIN. The PIN transmitted through the card reader terminal is compared by the smart card with the key 5 stored on the card. If they match, security state DF04 is entered. Security state DF04 allows read and write access to all data files. In state DF04 a locked user PIN, key 2, can be unlocked by successful verification of the personal unlock key (PUK) stored on the smart card as key 6. With this unlock procedure key 2 is changed and its error counter reset to 3. A successful PUK procedure initiates the transition from security state DF04 to DF03, i.e., the administrator mode is changed to user mode after successful card holder verification.

4.2 Interfaces

In the following discussion, we describe the model of the card that we used for generating test cases. Its interface and structure are shown in Fig. 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 discussion, however, we restrict ourselves to giving an abstracted version of the actual card that, after suitable transformations, was also used for the computation of test cases for the actual hardware. This requires translating the abstract commands into byte strings that the chip card processor's API is able to process. This step also includes inserting correct (or, depending on the test case, incorrect) PINs. In this case study, we concentrated on testing the actual implementation.

Note that we consider the use of fairly abstract behavior models as one of the keys to successful model-based testing. Models are supposed to be simple, in order to be understandable, and in order to be amenable to automatic test case generation. Missing information is in-

serted by means of driver components. Complexity is thus distributed between the model and the driver component.

For instance, we used the following – manually identified – 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 (the terminal) 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. Hence the modeling task included 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 is 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`.

4.3 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 also has a superuser mode (with a Personal Unblocking Key, or PUK). Authentication is achieved between card reader and (superuser) 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 Fig. 2.

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 the 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 control states (called "authentication" states in the requirements specification) and transitions between them. Some transitions (for instance, the reset transition)

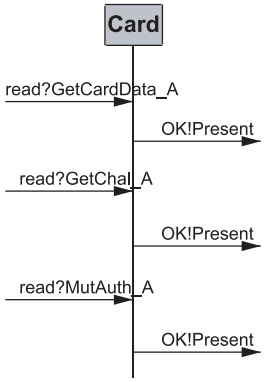

Fig. 2. Authentication sequence (simplified)

change the main state directly, whereas the authentication process requires two intermediate states between the connected authentication states (the "challenge" – random numbers – and some data have to be transferred).

In principle, these intermediate states could be modeled as control states (ovals) in the state transition diagram. However, this leads to a large number of control states and to many reset transitions (for each intermediate state). This is why we chose to encode these control states as data states that are changed by the postconditions of the respective transitions ([51] contains a description and comparison of both approaches to modeling). The intermediate states are encoded into a variable `AS:AuthState` of the type

```
data AuthState = Ready | GotData | GotChallenge
```

(cf. Fig. 3). That is to say, control states are encoded as data states. This reduces the number of control 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 (Fig. 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 commands 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 admitted (described by the equivalence class `ReadBin_A`), whereas updating results in an error (command equiva-

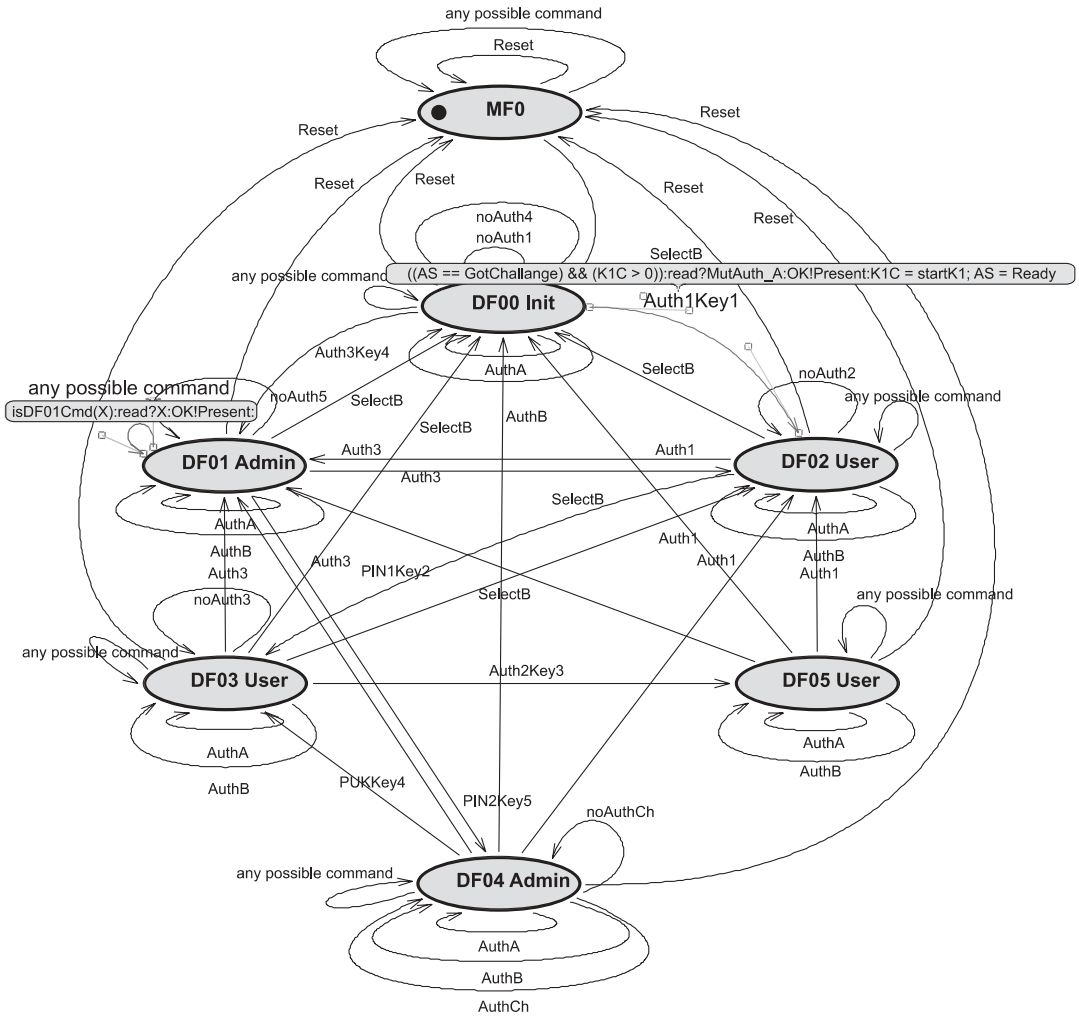

Fig. 3. Inhouse card: authentication with data states

lence class `UpdateBin_E`). For every state such a predicate is defined. This allows a flexible modeling process since every command can be allowed or excluded explicitly without changing the layout and the transitions of the STD.

5 Test case generation

In this section, we describe our approach to test case generation with symbolic execution on the grounds of Constraint Logic Programming. We restrict ourselves to a coarse description of the basic ideas; details of the translation [36, 37] as well as the embedding in an incremental development process [48] have been discussed elsewhere.

It turns out that test case generation for functional and structural test case specifications boils down to finding states in the model's state space (elsewhere [47], it is shown how test cases can be generated that satisfy the MC/DC coverage criterion. The idea is to first generate a set of test case specifications that enforce certain variable valuations and then generate test cases for them. The variable valuation is nothing but a state). The aim of symbolic execution of a model is then to find a trace – a test case – that leads to the specified state. Since it is in general impossible to completely test universal properties (properties that can be proved only with a possibly infinite set of infinite witnesses) at the level of the implementation, we take it for granted that the test engineer has transformed universal properties into such existential – "find state σ" – properties.

5.1 Test case generation

The basic idea behind our algorithm is a symbolic execution of the model. To this end, we generate a set of predicates, P_K, for each automaton, K. Recall that automata only occur in bottom-level components. Each predicate in P_K encodes one transition, and its arguments thus contain 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.

In a nutshell, each transition of a bottom-level component K – equipped with a state machine – is translated into a formula (a CLP program)

$$step^K(\vec{\sigma}_{src}, \vec{\iota}, \vec{o}, \vec{\sigma}_{dst}) \Leftarrow guard(\vec{\iota}, \vec{\sigma}_{src}) \wedge assgmt(\vec{o}, \vec{\sigma}_{dst}) \quad (1)$$

indicating that, given input $\vec{\iota}$, the component may proceed from control and data state $\vec{\sigma}_{src}$ to $\vec{\sigma}_{dst}$ by outputting $\vec{o}$, provided the transition's guard holds true. The successor state is determined by the transition arrow's destination and an assignment that updates the component's data state (the postcondition). Guards and assignments may contain arbitrary constraints. All these transition predicates form the set P_K.

Functional expressions in guards and assignments are easily translated into respective predicates. These predicates cannot directly deal with nested function calls, which is why they must be flattened. In essence, this is what happens when, say, nested arithmetic expressions are compiled into assembly language. Since the predicates allow recursive calls, recursion can be translated directly.

AutoFocus state machines are input enabled with an idling semantics. That is to say, if no transition from a given state can fire for a given output, then the system remains in its current state. For code generation purpose, this is easily implemented by means of an "else" case. For test case generation with Prolog, this idling condition must be computed explicitly. Even though there are some interesting consequences with respect to the use of negation in CLP, we omit its translation here.

5.1.1 Composition

When composing a set of components, $\mathcal{C}$, by putting them in a new SSD and connecting their ports, a driver predicate 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. Components K that are not leaves of the component hierarchy and thus consist of subcomponents $k_1, \ldots, k_n$ then recursively translate into

$$step^K(\vec{\sigma}^K_{src}, \vec{\iota}^K, \vec{o}^K, \vec{\sigma}^K_{dst}) \Leftarrow \bigwedge_{j=1}^{n} step^{k_j}(\vec{\sigma}^{k_j}_{src}, \vec{\iota}^{k_j}, \vec{o}^{k_j}, \vec{\sigma}^{k_j}_{dst}), \quad (2)$$

where internal channels, i.e., channels that connect subcomponents, are encoded as local variables of K and become parts of σ^K_{src} and σ^K_{dst}.

This ensures that the interface of the driver predicate $step^K$ 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 (recall 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 the involved system's components; and since there may be more than one input/output channel or local variable, these again usually are tuples.

This simple translation scheme only works because of the simple time-synchronous communication semantics of AutoFocus. When necessary, we model asynchronous communication by explicit buffer components.

5.1.2 Execution and search

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 [48], and its determination is an important task (which we do not consider further here). The "granularity" of the symbolic execution can vary. For instance, the computation of a guard can be performed by actually calling the predicates that correspond to its flat translation; it can, however, also be delayed.

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 of the underlying CLP engine 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, then binds this variable to a concrete value.[2] 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

[2] Choosing a good ordering for the transitions gives rise to different search strategies like best-first on the grounds of appropriate fitness functions [46].

were written down. This may result in loops because previously visited states can be visited again. In the extreme case, if 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. The problem becomes obvious if traces of a length of 10 000 or more are taken into account. We implemented three 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 reentered, another transition is chosen. In this sense, transitions are "interleaved" over time. 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 Cleanroom Reference Model [52], 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. The third approach relies on storing states and preventing hitherto visited states from being visited again (for an unlimited length of the generated traces, this does not result in incompleteness because of backtracking). This is described at the end of the next paragraph.

5.1.3 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 had been 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 the particular trace will turn out to be 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 is that obviously something went wrong, and backtracking is to be performed. The good news is 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 so common in Logic programming.

In order to use constraints for pruning the search tree, some additional steps have to be taken. Since AutoFocus allows for the 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 [22] – a metalanguage 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 recursive calls in order to avoid infinite loops. Lazy evaluation is achieved by means of (delaying) constraints. As mentioned above, the "granularity" of symbolic execution can vary: constraints can be restricted to arithmetic comparisons, but they can also contain arbitrary delayed function calls.

5.1.4 Storing sets of states

Constraints are used not only for representing sets of I/O but also for space-efficient storage of sets of states where previously visited states cannot be visited a second time. Since test case generation relies on symbolic execution where sets of states rather than single states are enumerated, it is thus necessary to check for inclusion of sets of states: if all ground instances of the representation of a set of states are included in the set of all ground instances of a previously stored representation of a set of states, then the further set may be ignored. This is because it is handled when the successors of the larger set are computed. From the point of view of validating the model, this is not problematic; from the point of view of testing an implementation, this means that traces involving states that have been visited more than once will not be executed. Unfortunately, in practice it often happens that exactly the same sequence of inputs does not yield an error if executed n times; the $n+1$-th time, however, an error occurs. The problem here is that models are abstractions of implementations, and one state of the model may well correspond to a multitude of states in the implementation. The problem can be alleviated by storing sequences of states rather than single states.

It turns out that checking for state inclusion is efficiently implementable in CLP. Let $\sigma = (\tilde{\sigma}, \mathcal{C}_\sigma)$ denote the representation of a set of states. $\tilde{\sigma}$ denotes the syntactic part, e.g., $\tilde{\sigma} = c(X)$ for a constructor c and a variable X, and $\mathcal{C}_\sigma$ denotes a set of constraints over the variables of σ, e.g., $\mathcal{C}_\sigma = \{X \in \mathbb{N} : X > 3\}$. $[\![\sigma]\!]$ then denotes the set of all ground instantiations of σ. In the example, this means that $[\![\sigma]\!] = \{c(4), c(5), \dots\}$. For technical reasons, we do not check for inclusion but rather for its negation. Let $\leq$ denote the usual term ordering; $mgu_{(\tilde{\nu},\tilde{\sigma})}$ denotes the most general unifier of two terms $\tilde{\nu}$ and $\tilde{\sigma}$. $\overline{\cdot}$ denotes the complement of its argument. Because of $[\![\nu]\!] \not\subseteq [\![\sigma]\!] \Leftrightarrow [\![\nu]\!] \cap \overline{[\![\sigma]\!]} \neq \emptyset$ for a new set of states, $[\![\nu]\!]$, and a previously visited set of states, $[\![\sigma]\!]$, it can be shown that $[\![\nu]\!]$ is not a specialization of $[\![\sigma]\!]$ (and thus must be stored) iff[3]

[3] We assume the cardinality of $[\![\nu]\!]$ to be greater than one.

$$\mathcal{C}_\nu \wedge \big(\tilde{\sigma} \leq \tilde{\nu} \Rightarrow mgu_{(\tilde{\nu},\tilde{\sigma})}(\overline{\mathcal{C}}_\sigma \wedge \mathcal{C}_\nu)\big),$$

which is trivial to implement in CLP. $\overline{\mathcal{C}}_\sigma$ denotes the negation of a set of constraints. For equalities and inequalities, this is syntactically easy to achieve (replace $=$ by $\neq$, $\geq$ by $<$, etc.). In essence, each visited set of states must be compared to all previously visited sets (represented as predicates; combination of sets into one single predicate is possible).

It is also possible not to check for the negation of inclusion but instead subtract the set of all previously visited states from the currently visited one. We omit this subject for the sake of brevity. The approach is conceptually similar to memoing [63], where multiple calls to the same predicate are prohibited. We do not memo at the level of predicates but rather at the level of explored (sets of) states, and we include general constraints in this scheme.

5.1.5 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, then, 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; this also shows how our approach extends to limit tests over time). Naturally, clever instantiations are a major problem in the generation of test cases.

5.1.6 Integration tests and compositionality

Our approach is compositional in that test cases for subsystems can be used to generate test cases for an integrated system [47]. This applies to (i) different components within one system, (ii) different increments of the model of a system, and (iii) different models – referring to the above example, this means that test cases for the chip card yield test cases for the terminal, and vice versa. In terms of different increments, automatic test generation technology naturally lends itself to its application in regression testing [48]. 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".

5.2 Discussion

At the level of exploring the model's state space, our approach is similar to explicit-state model checking. We do, however, compute with sets of states rather than single states, as is done in Spin or Murϕ. The relationship between (C)LP and model checking of (infinite) systems has been the subject of recent work [5, 11, 12, 21]. While the intention of 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 counterexamples.

Methodologically, there is a fundamental difference between testing and model checking. First, model checking is concerned with models, and testing is usually concerned with implementation. In order to check, for instance, assumptions on the environment, testing *must* complement technologies that rely on models only.

As a consequence, model checking allows the assessment of infinite sets of model traces in finite time. Actual implementations of embedded systems, however, must be verified or validated by means of finite sets of finite traces. The problem is that in general a, say, invariance property cannot be validated by a finite trace: witnesses of such a property are transition systems that usually encode infinitely many infinite traces. It is thus necessary to approximate such universal properties by properties that can be proved by finite witnesses (in CTL terminology, *EF* properties). This is why our test case generator is not equipped with mechanisms for checking universal properties (e.g., double stacks for liveness): we consider *EF* properties as the starting point for test case generation. It turns out that coverage-based test criteria as well as scenarios are expressible in terms of them [47].

The use of constraint languages is also crucial in the intended interactivity that we consider the key to scalability of our method: simplifying 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: other work contains an example where changing the maximum length by as little as 20 results in a change of as many as four orders of magnitude in terms of time needed to compute the specified test case [48].

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. We have shown elsewhere how the definition of fitness function can be used to direct the search that, for certain

applications, turns out to yield considerable gains in performance [49]. One application of this scheme is discussed in the next section.

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.

5.3 Application

We generated several test cases for this example that, after the necessary concretization, were fed into the actual hardware. The objective was to derive test sequences for several coverage criteria (states, transitions) as well as for functional purposes. Memory requirements for the generation of all test cases was smaller than 10 MB; we omit the details. All measurements were performed on a Pentium III/850 MHz, 512 MB RAM.

One of the test purposes was to achieve control state coverage. Our system computed the corresponding test cases in < 0.01 s 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 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 interleaving of transitions explained in Sect. 5.1.2). The first sequence took 7.8 s to compute; subsequent sequences could be obtained immediately.

Table 1 shows some experimental data for functional tests that required the counters to reach zero (the meaning of the asterisk is explained below).

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 h. 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 that we did not find the test case.

Table 1. Required time

cnt #	from–to	max steps	time [s]
2	3–0	11	8.8
3	14–12	15	41.3
3	14–0*	10	.52
4	14–12	15	132.4
4	14–0*	43	.345
6	15–14	8	84.96
6	15–0*	20	.1

5.3.1 Manual intervention

In the 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 control state, i.e., oval (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 an asterisk ($*$) indicate that we helped the system in the described manner. In the following section, we show how storing states makes the automatic computation of these test cases possible.

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 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 no 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.

5.3.2 Automatic generation

In the following discussion, we show the fully (and almost instantaneous) automatic determination of the test cases in Table 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 storing sets of already visited states (cf. Sect. 5.1.4). These results are taken from earlier work [46].

Best-first. As we will argue quantitatively in the following discussion, the ordering in which transitions from a given (control) state are taken is crucial for the performance of the test case generation. This issue becomes increasingly important when there are many transitions from that state. In some cases, it is possible to define a fitness function that with respect to a test case specification computes the (approximately) best transition to be taken. In the case of control states to be reached

or transitions to be taken, this fitness function is comparatively easy to define: from each control state c we compute the shortest path p_c to reach the desired destination state s. We then implement a best-first search by defining the transition ordering for state c as follows. Transitions that connect c with the second state in p_c are tried first. For the remaining transitions emanating from c, we choose those that lead to a state d satisfying the requirement that there be no d' where the length of $p_{d'}$ is strictly shorter than that of p_d.

Iterating this procedure leads to a transition ordering that first tries transitions from c that are on p_c. It then tries those transitions that lead to states with minimum shortest paths to s, then those that lead to states with the second minimum paths, etc. This algorithm works because each subpath of an optimal path is itself optimal. The advantage of this procedure is that best-first strategies based on the Euclidean distance between control states can be encoded statically. In [49] we present a more general approach that takes into account arbitrary distance measures on the control *and data* states that are computed at runtime.

In our example, we are interested in optimal orderings with respect to reaching state DF00Init for decrementing cnt_4 and to reaching DF04Admin in order to decrement cnt_6. This is because transition noAuth4 is responsible for decrementing cnt_4 and transition noAuthCH is responsible for decrementing cnt_6. We thus use the algorithm for reaching states as one that is capable of finding good transition orderings for reaching transitions; as a further step, we move transitions noAuth4 and noAuthCH in front of the respective transition sequences.

Note that this simple heuristics is, in general, applicable only to control states, as the number of data states may be too large. When the system contains data states, it is only a heuristic. This is the case in our example.

Breadth-first. The desire to get the shortest possible traces makes breadth-first search enter the game as it guarantees traces of minimum length. However, breadth-first search severely suffers from memory explosion (at least if we do not employ symbolic representations such as BDDs). For the sake of a smaller memory allocation, we did not store the traces in the experiment. It is noteworthy that the architecture of our system does not necessitate the implementation of a naive metainterpreter, as is usually done in breadth-first implementations in Prolog. We do not provide any numbers here [46].

Experiments. We consider two functional test case specifications, namely, decrementing counters cnt_4 and cnt_6.

Tables 2 and 3 summarize the results where sets of states are not visited twice. The following strategies were used: left-to-right ($\rightarrow$), randomized processes (rP) and randomized threads (rT). $\rightarrow$ is a strategy where the transitions that emanate from a control state are

Table 2. Search strategies for $cnt_4 \rightarrow 0$

depth	strat.	time	stet sets	memory
150	$\rightarrow$	.5-.8-178	46-333-9470	185-512-14546
	$\rightarrow$: σ/μ	53/22	2852/1505	4381/2312
	rP	.8-102-4724	232-7405-52403	356-11374-80491
	rP:σ/μ	1798/1298	20286/18539	31180/28446
	rT	2.6-153-621	653-8773-17693	1003-13476-27177
	rT:σ/μ	264/336	6849/10918	16771/10520
300	$\rightarrow$	.5-.6-377	56-79-14440	86-122-22180
	$\rightarrow$: σ/μ	113/38	4297/1552	6560/2383
	rP	1-2.9-2432	354-956-37282	544-1467-57265
	rP:σ/μ	915/534	14284/10035	21941/15413
	rT	1.5-1.7-93	330-398-6907	507-612-10609
	rT:σ/μ	30/16	2253/1570	3460/2411
400	$\rightarrow$	.6-.7-4706	42-112-46366	65-172-71218
	$\rightarrow$: σ/μ	1111/371	13853/4808	21278/7385
	rP	1-2.9-2432	354-956-37282	544-1469-57265
	rP:σ/μ	915/534	14284/10	21941/15414
	rT	1.3-1.7-3.3	254-386-945	390-593-1451
	rT:σ/μ	.7/265	229/668	352/404
500	$\rightarrow$	.6-.7-2900	70-140-41984	108-215-64488
	$\rightarrow$: σ/μ	870/291	12527/4403	19242/6764
	rP	1-1.2-3083	410-471-43107	630-724-66213
	rP:σ/μ	924/378	13434/6667	20635/10240
	rT	1.4-1.7-285	243-444-12154	373-682-18669
	rT:σ/μ	1.4/30.2	243/1640	373/2519

Table 3. Search strategies for $cnt_6 \to 0$

depth	strat.	time	stet sets	memory
50	$\to$ (7)	13.4-72.3-269	2485-6349-12445	3817-9752-19116
	$\to$: σ/μ	80/85	3600/5686	4731/9323
	rP (5)	58-242-346	5282-11578-13909	5282-11578-13909
	rP:σ/μ	95/219	2962/13909	4551/21365
	rT (4)	52-100-389	5072-7044-13756	7790-10033-21129
	rT:σ/μ	149/224	3824/9815	6024/14880
100	$\to$ (8)	.6-2.6-1026	23-745-24587	35-1145-37766
	$\to$: σ/μ	359/202	8861/6133	13611/9420
	rP (9)	19-1057-2124	2955-24851-35255	4539-38171-54151
	rP:σ/μ	827/930	12880/19192	19783/29479
	rT (8)	30-96-1173	4091-7150-23534	6284-10982-36148
	rT:σ/μ	402/319	6662/10445	16044/10233
150	$\to$	.5-64-1498	47-5803-28363	72-8914-43566
	$\to$: σ/μ	442/274	8385/8920	12880/13701
	rP	3.9-317-1714	1156-12643-31422	1775-19419-48264
	rP:σ/μ	620/656	11536/15734	17719/24167
	rT	3.2-145-1113	928-8816-22928	1426-13541-35218
	rT:σ/μ	318/267	6345/9656	9747/14832
200	$\to$	.9-70.5-685	349-6046-20271	536-9287-31137
	$\to$: σ/μ	271/285	7487/10439	11500/16035
	rP	.9-10.4-196	357-2179-10513	549-3347-16148
	rP:σ/μ	64/48	3268/3888	5020/5973
	rT	3.4-129-1057	1002-8229-22532	1539-12640-3904776
	rT:σ/μ	413/355	7949/1.02E04	1.5E06/665951
300	$\to$	.7-46.3-9122	244-4622-59001	375-7099-111096
	$\to$: σ/μ	2765/1353	17677/12686	32968/22107
	rP	96-1225-2314	7074-25550-35256	10866-39245-54153
	rP:σ/μ	701/1287	8894/24983	13660/38374
	rT	22-30-427	3354-3699-15094	5152-5682-23185
	rT:σ/μ	165/149	4724/7352	7257/11294

tried in an arbitrary but fixed order. rP denotes a strategy where ten processes are run in parallel and computation stops when the first process has succeeded. rT denotes a strategy where ten threads run in parallel and where the set of stored states is shared by all threads. $\to$ is different from the other strategies in that the ordering in which transitions are tried is always the same (i.e., one random ordering is fixed at the beginning of the generation procedure). The other strategies use a new random ordering whenever a state is visited. For each search depth, all experiments were conducted ten times.

Hyphen-separated triples include minimum, mean, and maximum values. σ and μ denote standard deviation and average values, respectively. We show the maximum search depth, the necessary time, the number of sets of states that were stored during execution, and the necessary memory. In the case of cnt_6, the number in brackets indicate the number of successful threads or processes. If no number is given, all ten runs were successful. Because of state storage, not all runs are successful.

One can see that rP and rT exhibit approximately the same characteristics. In general, the worst-case performance of rT is better than that of rP. In terms of the average case, rT tends to perform slightly better.

Somewhat to our surprise, the best strategy for this application is $\to$. An analysis of the visited sets of states revealed why. It is bound to the structure of the search problem. Good results (little time, little memory) are achieved if few control states (ovals in the state machines) are visited. The probability of getting a good result is rather high for many permutations of the transition orderings. If good transition orderings are chosen once, at the beginning, the result is good. If a new ordering is chosen within every step, then the probability of having to visit a further control state increases. If this happens, then the counter associated with this control state has to be decremented that results in many new global states.

In all cases, using directed search yields the results within fractions of a second.

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. For a discussion of the relationship of (bounded) model checking and testing, we refer to earlier work [46]; 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) s 112464 (73145) BDD nodes, and 3 (2.5) MB storage. Bounded model checking [66] with SATO fails to find these examples. Our approach with best-first search (see above, [46]) succeeds in finding the test case in less than 0.01 s. In the aforementioned subsequent studies [43], the models were too complex to be amenable to model checking.

6 Conclusion

Our understanding of model-based testing boils down to using abstract models for the generation of test cases. Their simplicity enables their intellectual mastery as well as automatic test case generation. In order to use traces of the model for testing an implementation, the missing information has to be inserted. This is done by means of driver components. Complexity is hence distributed between model and driver.

Test case generation relies on the use of selection criteria. These must be provided by a test engineer, and they represent "interesting" situations. Structural criteria exhibit the advantage that test cases can be generated fully automatically without further specifying the selection criterion. However, there is widespread agreement that they must be complemented by functional tests.

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 one to compute relevant test cases for industrial applications. This alleviates the tedious task of test development. In fact, a subsequent study [43] confirmed this impression. In the remainder of the paper, we briefly assess both the modeling capabilities of AutoFocus and the test generator.

Modeling. The 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 industrial testing. In addition, the integration of the modeling and the testing tools was identified as being 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. This is why, as a complement to functional test cases, automatically generating test cases from structural criteria is considered desirable.

We do not assess the quality of the generated tests here. This is because generally accepted notions of what constitutes a "good" test case do not exist. That is to say, test cases are as good as the respective test case specifications. In a subsequent study [43], we deliberately generated test cases without testing experience of the domain experts. Test case specifications were then structural, functional, and stochastic. The domain experts decided that the generated test cases "covered" (with an intutitive understanding of one test case "covering" another) their manual test cases. The advantage of automatically generated test cases is that once a model is built, they are cheap to generate, and test cases can be executed in parallel. The number of test cases is then restricted by the number of existing card reader terminals.

The model in this paper consists of just one component. Experience has shown that the approach is also applicable to more complex systems such as further chip card applications [43, 47] or the NetworkMaster of multimedia buses in modern vehicles [44, 45]. We are also optimistic that it is usable in the realm of mixed discrete-continuous systems [26]. In particular, it is possible to make use of test cases of a subsystem to generate test cases for a set of composed components. This issue of compositional test case generation – up to integration tests for the overall system – is discussed elsewhere [47]. The cited article shows how unit tests that satisfy a structural coverage criterion (modified decision/condition coverage, MC/DC) can be used to automatically generate test cases that satisfy MC/DC at the level of the integrated system.

The main problem in testing – as in formal verification – is the question of what an "interesting" test case (or a "relevant" property) is. Structural coverage criteria are advantageous in that they are independent of a particular application domain, but there is broad agreement that structural test cases should only be used as a complement to functional, application-specific tests. We consider domain-specific error classifications and checklists to be a promising first step toward the goal of a structured testing process.

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 remodeled in Lustre, and that existing test tools such as Lutess [16] or Gatel [38] 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. Ammann P, Black P, Majurski W (1998) Using model checking to generate tests from specifications. In: Proceedings of the 2nd IEEE international conference on formal engineering methods, Brisbane, Queensland, Australia, 9–11 December 1998, pp 46–54
2. Binder R (2001) Testing object-oriented systems: models, patterns, and tools. Addison-Wesley, Reading, MA
3. Bourhfir C, Dssouli R, Aboulhamid E (1996) Automatic test generation for EFSM-based systems. Technical Report IRO 1043, University of Montreal, August 1996
4. Brooks F (1986) No silver bullet. In: Proceedings of the 10th IFIP world computing conference, Dublin, Ireland, 1–5 September 1986, pp 1069–1076
5. Bultan T (1998) Automated symbolic analysis of reactive systems. PhD thesis, University of Maryland, College Park, MD
6. Burton S, Clark J, McDermid J (2001) Automatic generation of tests from Statechart specifications. In: Proceedings of the conference on formal approaches to testing of software, Aalborg, Denmark, 25 August 2001, pp 31–46
7. Chow T (1978) Testing software design modeled by finite-state machines. IEEE Trans Softw Eng SE-4(3):178–187
8. Ciarlini A, Frühwirth T (1999) Using Constraint Logic Programming for software validation. In: Proceedings of the 5th workshop on the German-Brazilian Bilateral Programme for Scientific and Technological Cooperation, Königswinter, Germany, March 1999
9. Claessen K, Hughes J (2000) QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the 5th ACM SIGPLAN international conference on functional programming, Montreal, 18–21 September 2000, pp 268–279
10. Clarke L (1976) A system to generate test data and symbolically execute programs. IEEE Trans Softw Eng SE-2(3): 215–222
11. Cui B, Dong Y, Du X, Kumar NK, Ramakrishnan C, Ramakrishnan I, Roychoudhury A, Smolka S, Warren D (1998) Logic programming and model checking. Lecture notes in computer science, vol 1490. Springer, Berlin Heidelberg New York, pp 1–20
12. Delzanno G, Podelski A (1999) Model checking in CLP. In: Proceedings of the 5th international conference on tools and algorithms for construction and analysis of systems, Amsterdam, 22–28 March 1999, pp 223–239
13. Denney R (1991) Test-case generation from Prolog-based specifications. IEEE Softw 8(2):49–57
14. De Vries R, Tretmans J, Belinfante A, Feenstra J, Feijs L, Mauw S, Goga N, Heerink L, de Heer A (2000) Côte de Resyste in Progress. In: Proceedings of Progress 2000 – workshop on embedded systems, Utrecht, The Netherlands, October 2000, pp 141–148
15. Du Bousquet L, Ouabdesselam F, Parissis I, Richier J-L, Zuanon N (2000) Specification-based testing of synchronous software. In: Proceedings of the 5th international workshop on formal methods for industrial critical systems, Berlin, 3–4 April 2000, pp 123–140
16. Du Bousquet L, Zuanon N (1999) An overview of Lutess, a specification-based tool for testing synchronous software. In: Proceedings of the 14th IEEE international conference on automated SW engineering, Cocoa Beach, FL, 12–15 October 1999, pp 208–215
17. Duran J, Ntafos S (1984) An evaluation of random testing. IEEE Trans Softw Eng SE-10(4):438–444
18. Edelkamp S, Lluch-Lafuente A, Leue S (2001) Directed Explicit Model Checking with HSF-SPIN. In: Proceedings of the 8th international SPIN workshop on model checking software, Toronto, 19–20 May 2001, pp 57–79
19. Fernandez J-C, Jard C, Jéron T, Viho C (1996) Using on-the-fly verification techniques for the generation of test suites. In: Proceedings of the 8th international conference on computer-aided verification, New Brunswick, NJ, 31 July–3 August 1996, pp 348–359
20. Frankl P, Weyuker E (1998) An applicable family of data flow testing criteria. IEEE Trans Softw Eng 14(10):1483–1498
21. Fribourg L (1999) Constraint logic programming applied to model checking. In: Proceedings of the 9th international workshop on logic-based program synthesis and transformation (LOPSTR'99), Venice, 22–24 September 1999. Lecture notes in computer science, vol 1817. Springer, Berlin Heidelberg New York, pp 30–41
22. Frühwirth T (1998) Theory and practice of constraint handling rules. J Logic Program 37(1–3):95–138
23. Goodenough J, Gerhart S (1975) Toward a theory of test data selection. IEEE Trans Softw Eng SE-1(2):156–173
24. Groce A, Visser W (2002) Model checking Java programs using structural heuristics. In: Proceedings of the international symposium on software testing and analysis, Rome, 22–24 July 2002, pp 12–21
25. Gutjahr W (1999) Partition testing versus random testing: the influence of uncertainty. IEEE Trans Softw Eng 25(5):661–674
26. Hahn G, Philipps J, Pretschner A, Stauner T (2003) Prototype-based tests for hybrid reactive systems. In: Proceedings of RSP'03, San Diego, 9–11 June 2003, pp 78–86
27. Hamlet D, Taylor R (1990) Partition test does not inspire confidence. IEEE Trans Softw Eng 16(12):1402–1411
28. Howden W (1975) Methodology for the generation of program test data. IEEE Trans Comput C-24(5):554–560
29. Howden W (1977) Symbolic testing and the DISSECT symbolic evaluation system. IEEE Trans Softw Eng SE-3(4):266–278
30. Howden W (1978) An evaluation of the effectiveness of symbolic testing. Softw Pract Exper 8:381–397
31. Huber F, Schätz B, Einert G (1997) Consistent graphical specification of distributed systems. In: Proceedings of the conference on industrial applications and strengthened foundations of formal methods (FME'97), Graz, Austria, 15–19 September 1997. Lecture notes in computer science, vol 1313. Springer, Berlin Heidelberg New York, pp 122–141
32. International Organization for Standardization (1995) International Standard ISO/IEC 7816: integrated circuit(s) cards with contacts
33. King J (1976) Symbolic execution and program testing. Commun ACM 19(7):385–394
34. Koch B, Grabowski J, Hogrefe D, Schmitt M (1998) AutoLink – a tool for automatic test generation from SDL specifications. In: Proceedings of the IEEE international workshop on industrial strength formal specification techniques, Boca Raton, FL, 20–23 October 1998, pp 114–127
35. Legeard B, Peureux F (2001) Génération de séquences de tests à partir d'une spécification B en PLC ensembliste. In: Proceedings of Approches Formelles dans l'Assistance au Développement de Logiciels, Nancy, France, June 2001, pp 113–130
36. Lötzbeyer H, Pretschner A (2000) AutoFocus on Constraint Logic Programming. In: Proceedings of (Constraint) Logic Programming and Software Engineering, London, 10 August 2000
37. Lötzbeyer H, Pretschner A (2000) Testing concurrent reactive systems with Constraint Logic Programming. In: Proceedings of the 2nd workshop on rule-based constraint reasoning and programming, Singapore, 22 September 2000
38. Marre B, Arnould A (2000) Test sequence generation from Lustre descriptions: GATEL. In: Proceedings of the 15th IEEE international conference on automated software engineering (ASE'00), Grenoble, France, 11–15 September 2000, pp 229-238
39. Meudec C (2000) ATGen: automatic test data generation using Constraint Logic Programming and Symbolic Execution. In: Proceedings of the 1st international workshop on automated program analysis, testing, and verification, Limerick, Ireland, 4–5 June 2000

40. Nielsen B (2000) Specification and test of real-time systems. PhD thesis, Department of Computer Science, Aalborg University, Aalborg, Denmark
41. Ntafos S (1988) A comparison of some structural testing strategies. IEEE Trans Softw Eng 14(6):868–874
42. Peleska J, Siegel M (1997) Test automation of safety-critical reactive systems. S Afric Comput J 19:53–77
43. Philipps J, Pretschner A, Slotosch O, Aiglstorfer E, Kriebel S, Scholl K (2003) Model-based test case generation for smart cards. In: Proceedings of FMICS'03, Trondheim, Norway, 5–7 June 2003, pp 168–182
44. Prenninger W, Pretschner A, Wagner S (2003) MOST NetworkMaster – AutoFocus model. Internal Study, BMW AG and TU München, Munich, Germany
45. Prenninger W, Pretschner A, Wagner S (2003) MOST NetworkMaster – generation of test harnesses. Internal Study, BMW AG and TU München, Munich, Germany
46. Pretschner A (2001) Classical search strategies for test case generation with Constraint Logic Programming. In: Proceedings of the workshop on formal approaches to testing of software, Aalborg, Denmark, August 2001, pp 47–60
47. Pretschner A (2003) Compositional generation for MC/DC test suites. In: Proceedings of TACoS'03, Warsaw, Poland, 13 April 2003, pp 1–11
48. Pretschner A, Lötzbeyer H, Philipps J (2001) Model based testing in evolutionary software development. In: Proceedings of the 11th IEEE international workshop on rapid system prototyping, Monterey, CA, 25–27 June 2001, pp 155–160
49. Pretschner A, Lötzbeyer H, Philipps J (2003) Model based testing in incremental system development. J Sys Softw 70(3):315–329
50. Pretschner A, Philipps J (2002) Szenarien modellbasierten Testens. Technical Report TUM-I0205, Institut für Informatik, Technische Universität München, Munich, Germany
51. Pretschner A, Slotosch O, Lötzbeyer H, Aiglstorfer E, Kriebel S (2001) Model based testing for real: the inhouse card case study. In: Proceedings of the 6th international workshop on formal methods for industrial critical systems, Paris, France, 16–17 July 2001, pp 79–94
52. Prowell S, Trammell C, Linger R, Poore J (1999) Cleanroom software engineering. Addison-Wesley, Reading, MA
53. Ramamoorthy C, Ho S, Chen W (1976) On the automated generation of program test data. IEEE Trans Softw Eng SE-2(4):293–300
54. Raymond P, Weber D, Nicollin X, Halbwachs N (1998) Automatic testing of reactive systems. In: Proceedings of the 19th IEEE symposium on real-time systems, Madrid, 2–4 December 1998, pp 200–209
55. Rusu V, du Bousquet L, Jéron T (2000) An approach to symbolic test generation. In: Proceedings of Integrated Formal Methods, Dagstuhl, Germany, 1–3 November 2000, pp 338–357
56. Sadeghipour S (1998) Testing cyclic software components of reactive systems on the basis of formal specifications. PhD thesis, Department of Informatics, TU Berlin
57. Tracey N (2000) A search-based automated test-data generation framework for safety-critical software. PhD thesis, Department of Computer Science, University of York, UK
58. Tretmans J (1996) Test generation with inputs, outputs and repetitive quiescence. Softw Concepts Tools 17(3):103–120
59. Ural H (1992) Formal methods for test sequence generation. Comput Commun 15(5):311–325
60. Vilkomir S, Bowen J (2001) Formalization of control-flow criteria of software testing. Technical Report SBU-CISM-01-01, South Bank University, London, UK
61. Visser W, Havelund K, Brat G, Park S (2000) Java PathFinder – second generation of a Java model checker. In: Proceedings of the workshop on advances in verification, Chicago, July 2000
62. Von Bochmann G, Petrenko A (1994) Protocol testing: review of methods and testing for software testing. In: Proceedings of the 1994 international symposium on software testing and analysis, Seattle, 17–19 August 1994, pp 109–124
63. Warren DS (1992) Memoing for logic programs. Commun ACM 35(3):93–111
64. Wegener J (2001) Evolutionärer Test des Zeitverhaltens von Realzeit-Systemen. PhD thesis, Humboldt Universität, Berlin
65. Weyuker E (1986) Axiomatizing software test data adequacy. IEEE Trans Softw Eng SE-12(12):1128–1138
66. Wimmel G, Lötzbeyer H, Pretschner A, Slotosch O (2000) Specification based test sequence generation with propositional logic. J Softw Test Validat Reliabil 10(4):229–248
67. Zhu H, Hall P, May J (1997) Software unit test coverage and adequacy. ACM Comput Surv 29(4):366–427