Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 191 (2007) 25–48

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Automatic Model-Based Generation of Parameterized Test Cases Using Data Abstraction

Jens R. Calamé[1] Natalia Ioustinova[2] Jaco van de Pol[3]

Dept. of Software Engineering
Centrum voor Wiskunde en Informatica
Amsterdam, The Netherlands

Abstract

Developing test suites is a costly and error-prone process. Model-based test generation tools facilitate this process by automatically generating test cases from system models. The applicability of these tools, however, depends on the size of the target systems.
Here, we propose an approach to generate test cases by combining data abstraction, enumerative test generation and constraint-solving. Given the concrete specification of a possibly infinite system, data abstraction allows to derive an abstract system, which is finite and thus suitable for the automatic generation of abstract test cases with enumerative tools. To execute abstract test cases, we have to instantiate them with concrete data. For data selection we make use of constraint-solving techniques.

Keywords: Conformance testing, model-based testing, test case generation, data abstraction, constraint-solving.

1 Introduction

Software failures can have expensive or dangerous consequences, so assuring the quality of a software product is very important. Software testing as a dynamic approach to validate a software product is widely accepted by academic and industrial communities. Depending on the view on an implementation under test (IUT), one differentiates between whitebox and blackbox testing. While there is knowledge about internal details of the IUT available for whitebox testing, this knowledge is absent in a blackbox test.

A test process involves the design of a test suite, its implementation and its execution. Each of these phases can be tedious and time-consuming for complex

[1] Email: jens.calame@cwi.nl

[2] Email: natalia.ioustinova@cwi.nl

[3] Email: vdpol@cwi.nl

doi:10.1016/j.entcs.2007.06.019

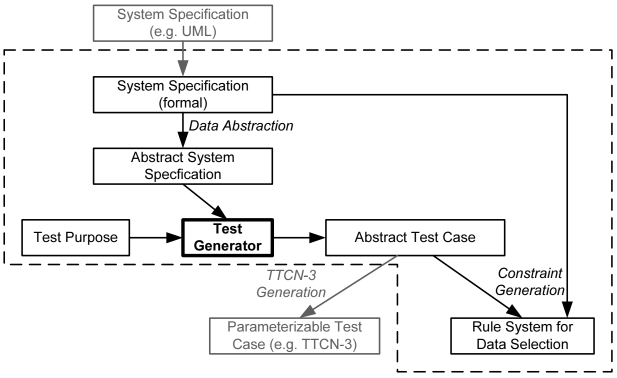
System Specification (e.g. UML)
System Specification (formal)
Data Abstraction
Abstract System Specfication
Test Purpose
Test Generator
Abstract Test Case
TTCN-3 Generation
Constraint Generation
Parameterizable Test Case (e.g. TTCN-3)
Rule System for Data Selection

Fig. 1. The test generation process

real-life systems. Therefore, there is ongoing research to automate each of these phases. In this paper, we provide an approach to automatic model-based test generation for blackbox testing.

Test cases can be generated either from a test model, i.e. a model of the test suite, or from a model of an IUT. In the first case, test cases are designed separately from the model of the IUT, for instance using frameworks like the UML 2 Testing Profile [27]. This leads to well-tailored test cases, but might also double the modeling effort during development. In the second case, the existing models of the IUT are examined by a test case generator. Model-based test generation is well-developed for conformance testing, which aims at checking whether an IUT conforms with its specification [22,33].

Existing model-based test generators usually rely on the enumeration of the state space of a specification. For open systems consisting of multiple components, communicating with each other and the system's environment, enumerative approaches lead to a large (sometimes infinite) number of test cases. Here, we propose an approach combining data abstraction, enumerative test generation and constraint-solving to solve this problem (see Figure 1).

Given the specification of an open system, we apply data abstraction which has already successfully been applied to model-checking open systems (e.g. [30]) to obtain a (finite) abstract system. From the abstracted system, we generate a set of abstract test cases. The generation of these test cases is guided by a test purpose [22], which defines the scenario of interest. The produced abstract test cases contain abstract data that should be concretized prior to test execution.

To instantiate abstract test cases with concrete data, we employ constraint-solving. In parallel to test generation and abstraction we transform the original specification into a rule system, which is used further for data selection. The test suite is transformed into a test oracle in order to retrieve test data. We use the data to instantiate the test case and execute it. While being executed, the IUT might diverge from the selected path. By online constraint-solving, we dynamically adapt the course of the test execution.

This paper is organized as follows: In the following subsection, we discuss papers that are related to our test generation approach. In Section 2, we give an overview of conformance testing. In Section 3, we describe the syntax and semantics of the specifications, we are working with. Preliminaries for constraint-solving are given in Section 4. In Section 5, we describe our approach for data abstraction, followed by the explanation of test generation in Section 6. We applied our approach to a case study, which is described in Section 7. Finally, we conclude with Section 8. Full proofs for the lemmata in this paper can be found in [4].

Related Work

The closest to our approach is *symbolic test generation* [8,14,23,29,36]. This method works directly on higher-level specifications given as Input-Output Symbolic Transition Systems without enumerating their state space. Given a test purpose and a specification, their product is built. Those works are founded on the *ioco* theory [32]. The coreachability analysis is in these cases over-approximated by Abstract Interpretation [10]. The concept of test generation with verification techniques (TGV, [22]) is also based on *ioco*.

The purpose and usage of abstraction techniques in our approach is conceptually different from the one of symbolic test generation, since we use data abstraction to avoid infinity caused by external data. This enables us to use existing enumerative test generation techniques to derive abstract test cases which are instantiated with concrete data derived by constraint-solving. In the symbolic test generation approach, approximate coreachability analysis is used to prune paths potentially not leading to Pass-verdicts. Both approaches are valid for any abstraction leading to an over-approximation of the IUT's behavior. They both employ constraint-solving to choose a single testing strategy during test execution. Which approach is actually more suitable for which class of systems can only be revealed by extensive use of the approaches.

Test data determination [17] for whitebox testing has been discussed in several recent papers [16,31,34]. They are mainly based on the technique of symbolic execution [9,24]. The constraint rule systems that we generate to determine test data, are comparable to [28], where also constraint rules are generated encoding the visible inputs and outputs, guards and internal state changes. These rules are used to generate a set of test cases by transforming a whole system specification into Prolog. However, test cases are already present in our case and the rule system is only needed to find concrete test data.

2 Testing Theory

Our approach is based on conformance testing that validates whether an implementation conforms to its specification. In a theory of conformance testing [32], the notion of conformance is formalized by a *conformance relation* between specification and implementation that are assumed to be input output labeled transition systems ($IOLTS$s). In this paper, we refer to a variant of the theory, described in [22]. Quiescence is not discussed here.

An $IOLTS$ is a tuple $(\Sigma, Lab, \rightarrow, \sigma_{init})$ with a non-empty set of states Σ, a dedicated initial state σ_{init}, an alphabet of action labels Lab and a transition relation $\rightarrow \subseteq \Sigma \times Lab \times \Sigma$. The set of labels Lab consists of three subsets of actions, Lab_I, Lab_O, and $\{\tau\}$ denoting input, output and internal actions. Input and output actions are *visible*, internal actions are *invisible*. An *IOLTS* is *deterministic* iff there is at most one outgoing transition for each action $\lambda \in Lab$ in each state $\sigma \in \Sigma$.

The behavior of an *IOLTS* M is given by sequences of states and transitions $\beta = \sigma_{init} \rightarrow \sigma_1 \rightarrow \ldots$ starting from the initial state. Traces are derived by projecting out the states, i.e. $[\![M]\!]_{trace} \subseteq Lab^{\star}$ is the set of traces of an *IOLTS* M. The relation **after** is defined as the set of states Σ', that can be reached if the system is in state σ and action λ is executed ($\Sigma' = \sigma$ **after** λ). It is also defined for an action λ possibly enabled after executing trace β ($\Sigma' = \beta$ **after** λ).

*IOLTS*s modeling *IUT*s are assumed to be *input-complete*, i.e. the implementation cannot refuse any input from the environment. Given a model M_{IUT} of an implementation and a model M_{Spec} of a specification, the implementation *conforms* to the specification iff for each trace β in $[\![M_{Spec}]\!]_{trace}$, M_{IUT} after this trace β produces only outputs that can be produced by M_{Spec} after β. In case, M_{Spec} is input complete, conformance is the standard trace inclusion relation [4].

We are interested in test generation where the test selection is guided by a *test purpose* [22]. A test purpose is a deterministic *IOLTS* M_{TP} that is equipped with a non-empty set of accepting states *Accept* and a set of refusing states *Refuse* which can be empty. Both accepting and refusing states are trap states, i.e. they cannot be left by any action anymore. Moreover, M_{TP} is complete in all the states except for the accepting and refusing ones. This means that in all states, all actions which are possible in M_{Spec} are enabled.

Assumption 1 (Treatment of Data in Test Purposes) *We assume that a test purpose M_{TP} is focused on the control flow of the described scenario only, so that the information about an action carried in the labels of Lab^{TP} is limited to the action names. Data parameters should not be subject of value assignment and are thus replaced by the* don't-care *parameter* $*$.

Test generation guided by a test purpose consists in building a standard *synchronous product* M_{SP} of M_{Spec} with M_{TP} and finally transforming it into a *complete test graph* M_{CTG} by assigning verdicts as possible results of test case execution (see Definition 2.2). The state space of the synchronous product M_{SP} forms the reachable part of $\Sigma^{Spec} \times \Sigma^{TP}$. The set $\rightarrow^{SP}$ is constructed by matching the transitions of M_{Spec} and M_{TP}. The set of accepting states is defined as $\Sigma_{acc} = \{(s, ACCEPT) | s \in \Sigma^{Spec}\}$.

Definition 2.1 (Complete Test Graph [22]) *A complete test graph CTG is an IOLTS* $M_{CTG} = (\Sigma^{CTG}, Lab^{CTG}, \rightarrow^{CTG}, \sigma^{CTG}_{init})$ *which is determined from the synchronous product* M_{SP} *in the following way:*

(i) *The set of actions is determined by mirroring the set of actions of* M_{CTG}*:*

[4] The difference with *ioco* [32] is that we do not abstract from τ-steps and that we do not yet consider quiescence.

$Lab^{CTG} = Lab_I^{CTG} \cup Lab_O^{CTG}$ *with* $Lab_O^{CTG} \subseteq Lab_I^{Spec}$ *and* $Lab_I^{CTG} = Lab_O^{Spec}$.

(ii) *The set of states is divided into four subsets* $\Sigma^{CTG} = \Sigma_{\mathsf{L2A}}^{CTG} \dot\cup \Sigma_{\mathsf{Inconc}}^{CTG} \dot\cup \Sigma_{\mathsf{Fail}}^{CTG}$ *and* $\Sigma_{\mathsf{Pass}}^{CTG} \subseteq \Sigma_{\mathsf{L2A}}^{CTG}$ *which are defined as follows:*

Lead to Accept: $\Sigma_{\mathsf{L2A}}^{CTG} = \{\sigma \in \Sigma^{SP} | \exists \beta \in [\![M_{SP}]\!]_{trace}(\sigma \rightarrow_\beta^{SP} \sigma' \wedge \sigma' \in \Sigma_{acc}^{SP})\}$,

Pass: *The set* $\Sigma_{\mathsf{Pass}}^{CTG} \subseteq \Sigma_{\mathsf{L2A}}^{CTG}$ *is defined as* $\Sigma_{\mathsf{Pass}}^{CTG} = \Sigma_{acc}^{SP}$. *This set may not be empty.*

Inconclusive: $\Sigma_{\mathsf{Inconc}}^{CTG} = \{\sigma' | \exists \sigma \in \Sigma_{\mathsf{L2A}}^{CTG}, \sigma' \notin \Sigma_{\mathsf{L2A}}^{CTG}, \lambda \in Lab_O^{SP}(\sigma \rightarrow_\lambda \sigma' \in \rightarrow^{SP})\}$,

Fail: $\Sigma_{\mathsf{Fail}}^{CTG} = \{\sigma_{\mathsf{Fail}}^{CTG}\}$, $\sigma_{\mathsf{Fail}}^{CTG} \notin \Sigma^{SP}$ *(implicit states).*

(iii) *The set of transitions of the CTG is defined as* $\rightarrow^{CTG} = \rightarrow_{\mathsf{L2A}}^{CTG} \cup \rightarrow_{\mathsf{Inconc}}^{CTG} \cup \rightarrow_{\mathsf{Fail}}^{CTG}$ *with:*

- $\rightarrow_{\mathsf{L2A}}^{CTG} = \rightarrow^{SP} \cap (\Sigma_{\mathsf{L2A}}^{CTG} \times Lab^{CTG} \times \Sigma_{\mathsf{L2A}}^{CTG})$,
- $\rightarrow_{\mathsf{Inconc}}^{CTG} = \rightarrow^{SP} \cap (\Sigma_{\mathsf{L2A}}^{CTG} \times Lab_I^{CTG} \times \Sigma_{\mathsf{Inconc}}^{CTG})$,
- $\rightarrow_{\mathsf{Fail}}^{CTG} = \{\sigma \rightarrow_\lambda \sigma_{\mathsf{Fail}}^{CTG} | \sigma \in \Sigma_{\mathsf{L2A}}^{CTG} \wedge \lambda \in Lab_I^{CTG} \wedge \sigma \textbf{ after } \lambda = \emptyset\}$.

The reason for mirroring inputs and outputs in the CTG lies in the relation between a test case and the IUT, as the input of the IUT is the output of the test case and vice versa. However, since a test case can normally not test all possible inputs of an IUT, its set of outputs Lab_O^{CTG} is limited to a subset of the IUT's set of inputs Lab_I^{Spec} by building the synchronous product of M_{Spec} and M_{TP}. The CTG may contain loops and choices between several outputs in the same state or between inputs and outputs. For this reason it is not controllable, i.e. the tester can for example not autoatically decide whether to expect an input from the IUT or to send an output to it.

The sets of accepting and refusing states of M_{SP} induce the sets of accepted and refused traces denoted $[\![M_{SP}]\!]_{atrace}$ and $[\![M_{SP}]\!]_{rtrace}$ respectively. $[\![M_{SP}]\!]_{atrace}$ are those traces that end in a state $\sigma \in \Sigma_{acc}^{SP}$ and $[\![M_{SP}]\!]_{rtrace} = [\![M_{Spec}]\!]_{trace} \setminus [\![M_{SP}]\!]_{atrace}$. Depending on the trace executed during the actual test, a verdict is assigned to assess the correctness of the IUT.

Definition 2.2 (Sound Verdict) *Predefined values for verdicts are:* Pass*,* Inconc*,* Fail *and* None*. The verdict is set by a function* setverdict : $[\![M_{SP}]\!]_{trace} \rightarrow$ Verdict *with:*

$$\text{setverdict}(\beta) = \begin{cases} \mathsf{Pass} & , \textit{iff } \beta \in [\![M_{SP}]\!]_{atrace} \wedge |\beta| > 0 \\ \mathsf{Inconc} & , \textit{iff } \beta \in [\![M_{SP}]\!]_{rtrace} \wedge |\beta| > 0 \\ \mathsf{Fail} & , \textit{iff } \beta \notin [\![M_{SP}]\!]_{atrace} \cup [\![M_{SP}]\!]_{rtrace} \wedge |\beta| > 0 \\ \mathsf{None} & , \textit{iff } |\beta| = 0 \end{cases}$$

The Pass verdict is assigned to those states of M_{CTG}, which correspond to the final states of traces from $[\![M_{SP}]\!]_{atrace}$ and thus to the *accept* states in the test purpose. The Inconc verdict is assigned to states from which accepting states are not reachable. In this case, the state is still on a trace of M_{Spec} but the trace does not satisfy the test purpose (traces from $[\![M_{SP}]\!]_{rtrace}$). All unspecified outputs lead

$$\begin{array}{ll}
\textbf{sort} & \mathit{Bool} \\
\textbf{func} & T :\rightarrow \mathit{Bool} \qquad F :\rightarrow \mathit{Bool} \\
\textbf{map} & \mathit{and} : \mathit{Bool} \times \mathit{Bool} \rightarrow \mathit{Bool} \\
\textbf{var} & b \colon \mathit{Bool} \\
\textbf{rew} & \mathit{and}(T, b) = b \qquad \mathit{and}(b, T) = b \qquad \mathit{and}(F, F) = F
\end{array}$$

Fig. 2. Data type for booleans

to the Fail verdict.

As we said before, M_{CTG} may contain choices between several outputs and choices between inputs and outputs. The test cases, we treat in this paper, are loopfree and controllable. A controllable test case M_{TC} is derived by resolving the choices mentioned, i.e. the test case does not contain these choices between outputs or between inputs and outputs anymore. A test case is executed in parallel with an IUT. A trace $\beta \in [\![M_{TC}]\!]_{\mathsf{Pass}}$ that leads to a Pass state is chosen. From this trace, several branches with one step lead to Inconc states, which represent traces in the test purpose ending in a refusing state.

Using test purposes as selection criteria, it is possible to generate test cases on-the-fly without generating the whole state space of a specification. However, a complete test graph can – due to all possible data – easily be infinite or at least too large to handle for enumerative techniques.

3 Syntax and Semantics of Specifications

In this section, we define the syntax and semantics of the systems we are working with. A *specification Spec* is given by a quintuple $\mathit{Spec} = (\mathit{Sort}, \mathit{Fun}, \mathit{Act}, \mathit{Comm}, \mathit{Proc})$ (for the specification language μCRL see [18]). It specifies an open system that communicates with its environment. *Sort* defines a set of data types for the declaration of variables. For each sort S there exists a set of constructors, which have the form $c :\rightarrow S$ or $c : S_1 \times \ldots \times S_n \rightarrow S$, resp., with $S_1, \ldots, S_n \in \mathit{Sort}$. These constructors are used to form values of sort S.

In *Fun*, functions of the form $f :\rightarrow S$ or $f : S_1 \times \ldots \times S_n \rightarrow S$, resp., are declared. Each of these functions is defined by one or more axioms on values of sorts $S_1, \ldots, S_n$. These axioms have the form $s = t$ where s and t are equally typed terms formed by any valid combination of typed variables and function symbols.

Figure 2 shows a sort *Bool* representing booleans (see **sort**), that is given by the two constructors T (for *true*) and F (for *false*, see **func**). The function *and* is declared in **map** with three axioms defining properties of *and* (see **rew**). Additionally, b is defined as a variable of sort *Bool* (see **var**).

The sets of actions *Act* and communicating actions *Comm* are necessary to declare the actions and communication issues necessary for the process definition. However, we will not discuss them here in detail. The process itself is defined as *Proc* in terms of *Linear Process Operators* [1]. To enable a graphical representation, however, we give a definition of processes based on the theory of Symbolic Tran-

$$\frac{l \longrightarrow_{?s(x)} \hat{l} \in Edg \qquad \forall v \in D}{(l,\eta) \longrightarrow_{?s(v)} (\hat{l}, \eta_{[x \mapsto v]})}\ \textsc{Input}$$

$$\frac{l \longrightarrow_{g \,\triangleright !s(e)} \hat{l} \in Edg \qquad [\![g]\!]_\eta = true \qquad [\![e]\!]_\eta = v}{(l,\eta) \longrightarrow_{!s(v)} (\hat{l}, \eta)}\ \textsc{Output}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, x:=e} \hat{l} \in Edg \qquad [\![g]\!]_\eta = true \qquad [\![e]\!]_\eta = v}{(l,\eta) \rightarrow_\tau (\hat{l}, \eta[x \mapsto v])}\ \textsc{Assign}$$

Table 1
Step semantics of process definition P ($Spec \to M$)

sition Systems. A process definition $Proc$ can thus be described by a four-tuple $(Var, Loc, \sigma_{init}, Edg)$, where Var denotes a finite set of variables, and Loc denotes a finite set of *locations* or control states. A mapping of variables to values is called a valuation; we denote the set of valuations by $Val = \{\eta \mid \eta\colon Var \to D\}$. Let $\Sigma = Loc \times Val$ be the set of states, where a process has one designated initial state $\sigma_{init} = (l_{init}, \eta_{init}) \in \Sigma$. The set $Edg \subseteq Loc \times Act \times Loc$ denotes the set of edges. An *edge* describes changes of configurations specified by an *action* from a set Act. Considering locations as nodes and edges as edges, such a specification can also be graphically represented as a Symbolic Transition System.

As actions, we distinguish (1) *input* of a signal s with a local variable to which a value can be assigned, (2) *output* of a signal s together with a value described by an expression, and (3) *assignments*. Every action except inputs is *guarded* by a boolean expression g, its guard. The three classes of actions are written as $?s(x)$, $g \triangleright !s(e)$, and $g \triangleright x := e$, respectively, and we use α, $\alpha' \ldots$ when leaving the class of actions unspecified. For an edge $(l, \alpha, \hat{l}) \in Edg$, we write more suggestively $l \longrightarrow_\alpha \hat{l}$.

The behavior of the process is then given by sequences of states $\zeta = \sigma_{init} \to \sigma_1 \to \ldots$ starting from the initial one. The step semantics is given by an *IOLTS* $M = (\Sigma, Lab, \to, \sigma_{init})$, where $\to\ \subseteq \Sigma \times Lab \times \Sigma$ is a labeled transition relation between states. The labels differentiate between internal τ-steps and communication steps, either input or output, which are labeled by a signal and a value being transmitted, i.e. $?s(v)$ or $!s(v)$, respectively. We assume that the set of signals coming from the environment and the set of signals exchanged within the system are disjoint.

The semantics is given by the inference rules in Table 1. Receiving a signal with a communication parameter x, $l \longrightarrow_{?s(x)} \hat{l} \in Edg$, results in an update of the valuation $\eta_{[x \mapsto v]}$ according to the parameter of the signal (rule Input). Output, $l \longrightarrow_{g \,\triangleright !s(e)} \hat{l} \in Edg$, is guarded, so sending a message involves evaluating the guard and the expression according to the current valuation. It leads to a change of location of the process that sends the message (rules Output). Assignments, $l \longrightarrow_{g \,\triangleright\, x:=e} \hat{l} \in Edg$, result in a change of the location and an update of the valuation $\eta_{[x \mapsto v]}$, where $[\![e]\!]_\eta = v$. Assignments are internal, so assignment transitions are labeled by τ (rule Assign).

Although we are working with specifications containing only one process definition, it does not limit our approach. The realization of our approach works on linear process operators [1]. For these, existing linearization techniques [19] can be used to obtain a single process definition for a parallel composition of a finite number of process definitions by eliminating communication and parallel composition.

4 Constraint-Solving Preliminaries

In this section we give an overview of notions related to constraint-solving [25].

A *constraint domain* $\mathcal{D}$ consists of a set of n-ary constraint symbols which describe relations and a logical theory $\mathcal{T}$. An example for such a constraint symbol is "$\leq$". A *primitive constraint* $c(X_1, \dots, X_n)$ is constructed from a constraint symbol and terms in the corresponding value set V_i for every argument position. An example for a primitive constraint is $\leq (X, Y)$ defining the relation $X \leq Y$.

A *constraint* is of the form $C = c_1 \wedge \dots \wedge c_m$ where $m \geq 0$ and $c_1, \dots, c_m$ are primitive constraints. We use *vars*(C) to denote the set of variables of constraint C. A *valuation* θ of a constraint C is a mapping of variables of *vars*(C) to values of $\langle V_1, \dots, V_n \rangle$ in $\mathcal{D}$. A logical theory $\mathcal{T}$ determines which constraints hold and which constraints do not hold under a certain valuation θ. If the constraint C holds for valuation θ under theory $\mathcal{T}$ of constraint domain $\mathcal{D}$, this is denoted $\mathcal{D} \models [\![C]\!]_\theta$. There are two distinct constraints *true* and *false* which behave the same for any theory. The tautology *true* always holds, while the contradiction *false* never holds.

Two problems are associated with C: the *solution problem* and the *satisfaction problem*. The first one determines a particular solution, the latter one determines whether there is at least one solution. Let θ be a valuation for C. θ is a *solution* for C if $[\![C]\!]_\theta$ holds, i.e. $\mathcal{D} \models [\![C]\!]_\theta$. A constraint C is *satisfiable* if it has at least one solution.

A *constraint solver solv*() for a constraint domain $\mathcal{D}$ is a decision procedure that takes as an input a constraint C and returns either *true*, *false* or *unknown*. Whenever *solv*(C) returns *true*, C is satisfiable. Whenever *solv*(C) returns *false*, there is no solution for C and C is unsatisfiable. The value *unknown* indicates that a solution might exist, but could not be determined.

A *user defined* constraint is of the form $p(t_1, \dots, t_n)$ where p is an n-ary predicate and $t_1, \dots, t_n$ are terms (variables, constants or functions) from a constraint domain. An example for a user-defined constraint is $a(X, f(Y))$, which takes two parameters (the variable X and the function $f(Y)$, with Y again being a variable). A *literal* is either a primitive constraint or a user defined constraint. A *rule* R, for instance in Prolog, is of the form $A \mathrel{:-} B$ where A is a user defined constraint and B is a sequence of literals. For example, $a(X, f(Y)) \mathrel{:-} X > 0, Y < X$ incorporates the two primitive constraints $X > 0$ and $Y < X$ in the sequence of literals. A *fact* is a rule with an empty sequence of literals, i.e. a rule of the form $A \mathrel{:-} \Box$, where $\Box$ is used to denote an empty sequence of literals. A *constraint logic program* $\mathcal{P}$ is a sequence of rules.

A *goal* or a *query* G is a sequence of literals, i.e. $G = L_1, \dots, L_m$ with $m > 0$. If $m = 0$, then G is an *empty* query denoted $\Box$. Let query G be of the form

$L_1, \ldots, L_{(i-1)}, L_i, L_{(i+1)}, \ldots, L_m$ and $L_i = p(s_1, \ldots, s_n)$. A state in a constraint logic program is given by a pair $\langle G \mid C \rangle$ with G being the actual goal and C being the constraint store, storing all relevant conditions for this state. The transitions between these states are *derivation steps.* A *derivation* [25] is a full trace from the initial to the final state, $\langle G_1 \mid C_1 \rangle \Rightarrow \ldots \Rightarrow \langle \Box \mid C \rangle$.

5 Data Abstraction

In this section, we describe the idea of data abstraction to close open systems for test generation. First, we explain the approach, before we discuss the relation between a concrete and an abstracted system.

We do not make any assumptions about the environment of an IUT, i.e. we take the most general one. Signals coming from the environment can thus have *any* value. This often boosts the state space of the system to infinity. We abstract data that is directly or indirectly influenced by the environment to one value *chaos*, denoted $\top$. Values that are not influenced by the environment remain the original ones, and so they should be treated in the same way as in the original system. This data abstraction was first proposed in [30] for model checking open systems. A system obtained by this approach is a safe abstraction of the original one, meaning, it shows *at least* the behavior of the original system [21,30].

We implement data abstraction as a transformation on the level of system specification. Abstraction on the level of specifications is well developed within the Abstract Interpretation framework [10,11,12]. The program transformation implementing this data abstraction transforms the signature and the process definition. For each sort S, we introduce a sort $S^\top$ that consists of two constructors, $\top_S :\to S^\top$ and $\kappa : S \to S^\top$. The first constructor defines a $\top$ value of the sort. The constructor κ (*known*) lifts values of sort S to values of sort $S^\top$. For each concrete mapping $m : S_1 \times \cdots \times S_n \to S_{n+1}$, we define a mapping $m^\top : S_1^\top \times \cdots \times S_n^\top \to S_{n+1}^\top$ mimicking the original one on the abstracted sorts. In the general case, mimicking is ensured by providing the following rewrite rules for each abstract mapping $m^\top$:

$$m^\top(\kappa(x_1), \ldots, \kappa(x_n)) = \kappa(m(x_1, \ldots, x_n))$$
$$m^\top(x_1, \ldots, x_n) = \top_{S_{n+1}} \text{ if } x_i \text{ is } \top_{S_i} \text{ for some } i \in \{1; ...; n\}$$

The transformation of the process specification consists in lifting all variables, expressions and guards to the new sorts. Each occurrence of a variable x of sort S, is substituted by a variable $x^\top$ of type $S^\top$ where $S^\top$ is a safe abstraction of sort S. Each occurrence of an expression e of type S is lifted to the expression $e^\top$ of sort $S^\top$. Thereby, all the newly introduced symbols (constructors and rewrite rules) are used and replace the original ones.

Transformation of guards is similar to the transformation of expressions. Every occurrence of a guard g is lifted to a guard $g^\top$ of type $Bool^\top$. While transforming guards we should ensure that the abstract system shows *at least* the behavior of the original system. Therefore, the guards valuated to $\kappa(true)$ or $\kappa(false)$ behave like

$$
\begin{array}{rl}
\textbf{sort} & \mathit{Bool}^{\mathbb{T}} \\
\textbf{func} & \mathbb{T}_{\mathit{Bool}} \colon\ \to \mathit{Bool}^{\mathbb{T}} \\
 & \kappa_{\mathit{Bool}} \colon \mathit{Bool} \to \mathit{Bool}^{\mathbb{T}} \\
\textbf{map} & \mathit{and}^{\mathbb{T}} : \mathit{Bool}^{\mathbb{T}} \times \mathit{Bool}^{\mathbb{T}} \to \mathit{Bool}^{\mathbb{T}} \\
 & \gamma : \mathit{Bool}^{\mathbb{T}} \to \mathit{Bool} \\
\textbf{var} & b, b' \colon \mathit{Bool} \\
\textbf{rew} & \mathit{and}^{\mathbb{T}}(\kappa(b), \kappa(b')) = \kappa(\mathit{and}(b, b')) \\
 & \mathit{and}^{\mathbb{T}}(\mathbb{T}_{\mathit{Bool}}, \kappa(F)) = \mathit{and}^{\mathbb{T}}(\kappa(F), \mathbb{T}_{\mathit{Bool}}) = \kappa(F) \\
 & \mathit{and}^{\mathbb{T}}(\mathbb{T}_{\mathit{Bool}}, \kappa(T)) = \mathit{and}^{\mathbb{T}}(\kappa(T), \mathbb{T}_{\mathit{Bool}}) = \mathbb{T}_{\mathit{Bool}} \\
 & \mathit{and}^{\mathbb{T}}(\mathbb{T}_{\mathit{Bool}}, \mathbb{T}_{\mathit{Bool}}) = \mathbb{T}_{\mathit{Bool}} \\
 & \gamma(\mathbb{T}_{\mathit{Bool}}) = T \\
 & \gamma(\kappa(b)) = b
\end{array}
$$

Fig. 3. Transformed sort $\mathit{Bool}^{\mathbb{T}}$

$$\frac{l \longrightarrow_{?s(x)} \hat{l} \in \mathit{Edg}}{l \longrightarrow_{?s(\mathbb{T})} \longrightarrow_{\mathit{true}\, \triangleright\, x:=\mathbb{T}} \hat{l} \in \mathit{Edg}^{\mathbb{T}}}\ \textsc{SInput}^{\mathbb{T}}$$

$$\frac{l \longrightarrow_{g\, \triangleright\, !s(e)} \hat{l} \in \mathit{Edg}}{l \longrightarrow_{\gamma(g^{\mathbb{T}})\, \triangleright\, !s(e^{\mathbb{T}})} \hat{l} \in \mathit{Edg}^{\mathbb{T}}}\ \textsc{SOutput}^{\mathbb{T}}$$

$$\frac{l \longrightarrow_{g\, \triangleright\, x:=e} \hat{l} \in \mathit{Edg}}{l \longrightarrow_{\gamma(g^{\mathbb{T}})\, \triangleright\, x:=e^{\mathbb{T}}} \hat{l} \in \mathit{Edg}^{\mathbb{T}}}\ \textsc{SAssign}^{\mathbb{T}}$$

Table 2
Transformation of edges ($\mathit{Spec} \to \mathit{Spec}^{\mathbb{T}}$)

guards evaluating to *true* or *false*, respectively. The guards valuated to $\mathbb{T}$ behave as guards evaluating to *true*. We implement this by introducing an extra mapping $\gamma\colon \mathit{Bool}^{\mathbb{T}} \to \mathit{Bool}$ that is *true* whenever a guard is evaluated either to $\mathbb{T}$ or to $\kappa(\mathit{true})$ and *false* otherwise. To avoid introducing unnecessary nondeterminism, we apply a more refined, technically speaking a non-strict, transformation to the sort *Bool*. Its abstraction, sort $\mathit{Bool}^{\mathbb{T}}$, is shown in Figure 3.

Definition 5.1 (May Semantics for Chaotic Guards) *While a guard g is defined as a function $g : \Sigma \to \{\mathit{true}, \mathit{false}\}$, a chaotic guard is defined as a function $g^{\mathbb{T}} : \Sigma^{\mathbb{T}} \to \{\mathit{true}, \mathit{false}, \mathbb{T}\}$. To map this three valued logic back to a two valued logic, a* may-*function $\gamma : \{\mathit{true}, \mathit{false}, \mathbb{T}\} \to \{\mathit{true}, \mathit{false}\}$ is defined as follows: $\gamma(\kappa(\mathit{true})) = \mathit{true}$, $\gamma(\kappa(\mathit{false})) = \mathit{false}$ and $\gamma(\mathbb{T}) = \mathit{true}$.*

After transforming the signature and lifting system variables, expressions and guards, we obtain a system that still can receive all possible values from the environment. The environment can influence data only via inputs. We transformed every input $l \longrightarrow_{?s(x)} \hat{l}$ from the environment into an input of signal s parameterized by the $\mathbb{T}$-value of the proper sort followed by assigning this $\mathbb{T}$-value to the variable x

$$\frac{l \longrightarrow_{?s(x)} \longrightarrow_{true \,\triangleright\, x:=\mathbb{T}} \hat{l} \in Edg^{\mathbb{T}}}{(l, \eta^{\mathbb{T}}) \longrightarrow_{?s(\mathbb{T})} (\hat{l}, \eta^{\mathbb{T}}_{[x \mapsto \mathbb{T}]})}\ \textsc{Input}^{\mathbb{T}}$$

$$\frac{l \longrightarrow_{\gamma(g^{\mathbb{T}}) \,\triangleright\, !s(e)} \hat{l} \qquad [\![\gamma(g^{\mathbb{T}})]\!]_{\eta^{\mathbb{T}}} = true \qquad [\![e^{\mathbb{T}}]\!]_{\eta^{\mathbb{T}}} = v}{(l, \eta^{\mathbb{T}}) \longrightarrow_{!s(v)} (\hat{l}, \eta^{\mathbb{T}})}\ \textsc{Output}^{\mathbb{T}}$$

$$\frac{l \longrightarrow_{\gamma(g^{\mathbb{T}}) \,\triangleright\, x:=e} \hat{l} \qquad [\![\gamma(g^{\mathbb{T}})]\!]_{\eta^{\mathbb{T}}} = true \qquad [\![e^{\mathbb{T}}]\!]_{\eta^{\mathbb{T}}} = v}{(l, \eta^{\mathbb{T}}) \rightarrow_{\tau} (\hat{l}, \eta^{\mathbb{T}}{}_{[x \mapsto v]})}\ \textsc{Assign}^{\mathbb{T}}$$

Table 3
Step-semantics of transformed edges ($Spec^{\mathbb{T}} \rightarrow M^{\mathbb{T}}$)

(see rule SInput$^{\mathbb{T}}$ in Table 2). Assignments and outputs are treated as explained before. The semantics of the transformed system are given by the inference rules in Table 3. Given an *IOLTS* M, we derive $M^{\mathbb{T}}$ by applying data abstraction on the specification of M.

$M^{\mathbb{T}}$ can receive only $\mathbb{T}$ values from environment, so the infinity of environmental data is collapsed into one value. Basically, the transformed system shows at least the traces of the original system where data influenced by environment are substituted by $\mathbb{T}$ values. This means, that $M^{\mathbb{T}}$ simulates M. This simulation relation is now defined for concrete and abstracted *IOLTS*s. It is not a standard relation, since we allow the abstraction of actions. Further, we give an overview of preservation results based on [21,20].

Definition 5.2 ($\leq$-Simulation) *Let* $M_1 = (\Sigma^1, Lab^1, \rightarrow^1, \sigma_0^1)$ *and* $M_2 = (\Sigma^2, Lab^2, \rightarrow^2, \sigma_0^2)$ *be two IOLTSs.* $(\leq_a, \leq_b)$ *with* $\leq_a \subseteq \Sigma^1 \times \Sigma^2$ *and* $\leq_b \subseteq Lab^1 \times Lab^2$ *is a simulation, iff* $\forall \sigma_1, \hat{\sigma}_1, \sigma_2, \lambda_1\ \exists \hat{\sigma}_2, \lambda_2 \big(\sigma_1 \leq_a \sigma_2 \wedge \sigma_1 \rightarrow_{\lambda_1} \hat{\sigma}_1 \Rightarrow (\lambda_1 \leq_b \lambda_2 \wedge \hat{\sigma}_1 \leq_a \hat{\sigma}_2 \wedge \sigma_2 \rightarrow_{\lambda_2} \hat{\sigma}_2)\big)$, *where* $\sigma_1, \hat{\sigma}_1 \in \Sigma^1, \lambda_1 \in Lab^1, \sigma_2, \hat{\sigma}_2 \in \Sigma^2, \lambda_2 \in Lab^2$.

We write $M_1 \preceq_{\leq} M_2$ *if there is such a relation between* M_1 *and* M_2*, also relating their initial states* $\sigma^1_{init} \leq_a \sigma^2_{init}$.

Before relating traces of the transformed system to the traces of the original system, we define an order relation on the states and on the labels of the systems. To relate states $Loc \times Val$ of the original system with the states of the transformed system $Loc \times Val^{\mathbb{T}}$, we define the relation $\leq_S$ as:

Definition 5.3 (Relation $\leq_S$) *Let* $\sigma = (l, \eta)$ *and* $\sigma^{\mathbb{T}} = (l', \eta^{\mathbb{T}})$ *be two states of the IOLTSs M and* $M^{\mathbb{T}}$ *with specifications Spec and* $Spec^{\mathbb{T}}$. $\leq_S$: $(Loc \times Val) \times (Loc \times Val^{\mathbb{T}})$ *is defined as* $\sigma \leq_S \sigma^{\mathbb{T}}$ *iff* $l = l' \wedge \forall x \in Var\big([\![x]\!]_{\eta^{\mathbb{T}}} = \mathbb{T} \vee [\![x]\!]_{\eta^{\mathbb{T}}} = \kappa([\![x]\!]_{\eta})\big)$.

To relate labels Lab of the original system with the labels of the transformed system $Lab^{\mathbb{T}}$, we define the relation $\leq_L$: $Lab \times Lab^{\mathbb{T}}$.

Definition 5.4 (Relation $\leq_L$) *Let* $\lambda \in Lab$ *and* $\lambda^{\mathbb{T}} \in Lab^{\mathbb{T}}$. *Then* $\lambda \leq_L \lambda^{\mathbb{T}}$ *is defined as follows:*

- $\tau \leq_L \tau$
- $?s(v) \leq_L ?s(v')$ *iff either* $v' = \mathbb{T}$ *or* $v' = \kappa(v)$

- $!s(v) \leq_L !s(v')$ *iff either* $v' = \top$ *or* $v' = \kappa(v)$

The following lemma states the simulation relation between M and $M^\top$ (see also the diagram below).

$$\begin{tikzcd} Spec \arrow[r, "Tab.\ 2"] \arrow[d, "Tab.\ 1"'] & Spec^\top \arrow[d, "Tab.\ 3"] \\ M \arrow[r, phantom, "\preceq_\leq"] & M^\top \end{tikzcd}$$

Lemma 5.5 *Let Spec be a specification and* $Spec^\top$ *be a specification obtained from Spec by the transformation defined in this section. Let* M *and* $M^\top$ *be IOLTSs obtained from respectively Spec and* $Spec^\top$ *by the rules in Table 1 or Table 3, respectively. Then* $M \preceq_\leq M^\top$ *and* $(\leq_S, \leq_L)$ *is this simulation.*

Proof sketch: The lemma is proven by checking the conditions for simulation in Definition 5.2 separately for input and output actions as well as for τ-steps. □

In the rest of the section, we lift this simulation relation on specifications to the inclusion of accepted and refused traces in the synchronous product of the abstracted system and the test purpose.

Lemma 5.6 *Let* M, N *be IOLTSs with* $M \preceq_\leq N$ *based on the simulation* $(\leq_S, \leq_L)$*. Let* P *be a test purpose. Then* $M \times P \preceq_{\leq'} N \times P$ *with* $(\leq_{S'}, \leq_L)$ *being this simulation. In this case holds:* $(\sigma^M, \sigma^P) \leq_{S'} (\sigma^N, \hat{\sigma}^P)$ *iff* $\sigma^M \leq_S \sigma^N$ *and* $\sigma^P = \hat{\sigma}^P$*. The relation* $\leq_L$ *is the same as defined in Definition 5.4.*

Proof sketch: The proof is analogous to that for Lemma 5.5, extended for the consideration of a synchronous product. □

Definition 5.7 (≤-inclusion on traces) *Let* ζ *and* ρ *be traces of IOLTSs* M_1 *and* M_2*. Trace* ρ *≤-includes* ζ*, written* $\zeta \leq \rho$*, iff* $|\zeta| = |\rho|$ *and* $\zeta_\lambda(i+1) \leq_L \rho_\lambda(i+1)$ *for all* $i \in \{0; \ldots; |\zeta|\}$*.*

Definition 5.8 (≤-inclusion on automata) *The set of traces generated by IOLTS* M_2 ≤-includes *the set of traces generated by IOLTS* M_1*, written as* $[\![M_1]\!]_{trace} \subseteq_\leq [\![M_2]\!]_{trace}$*, iff for every trace* ζ *of* M_1 *there exists a trace* ρ *in* M_2 *such that* $\zeta \leq \rho$*.*

Lemma 5.9 *Let TP be a test purpose,* M_{SP} *be a synchronous product of* M *with* M_{TP}*, and* $M_{SP}^\top$ *be a synchronous product of* $M^\top$ *with* M_{TP}*. Then* $[\![M_{SP}]\!]_{atrace} \subseteq_\leq [\![M_{SP}^\top]\!]_{atrace}$ *and* $[\![M_{SP}]\!]_{rtrace} \subseteq_\leq [\![M_{SP}^\top]\!]_{rtrace}$*.*

Proof sketch: To prove the lemma, we have to take Assumption 1 into consideration. We first show that the set of accepted and refused traces of M_{SP} in isolation is a subset of the union of accepted and refused traces in $M_{SP}^\top$. Then it is shown that the accepted traces of M_{SP} and the refused traces of $M_{SP}^\top$ do not have a common intersection. The same is shown for the refused traces of M_{SP} and the accepted traces of $M_{SP}^\top$. □

6 Testing with Abstractions

In this section, we describe the approach of test selection and execution with data abstraction. First, we give an algorithmic overview of the whole process. Then, we describe how the necessary rule system is built and how test selection and execution work. Finally, we review our approach and prove the soundness of verdicts assigned to a test execution.

Test Process Overview

Algorithm 6.1 (**SelectAndExecTest**($Spec$, TP) : $verdict \in \{\mathsf{None}, \mathsf{Pass}, \mathsf{Inconc}, \mathsf{Fail}\}$)

```
1  setVerdict(None);
2  $Spec^{\top} := abstract(Spec)$;
3  $\mathcal{RS} := buildRuleSystem(Spec)$;
4  $M^{\top}_{Spec} := generateLTS(Spec^{\top})$;
5  $M_{TP} := generateLTS(TP)$;
6  $M^{\top}_{CTG} := generateCTG(M^{\top}_{Spec}, M_{TP})$;
7  $M^{\top}_{TC} := selectATC(M^{\top}_{CTG})$;
8  while $M^{\top}_{TC} \neq no\_testcase$
9    $(\beta, \theta) := NewPassTrace(no\_trace, \emptyset, M^{\top}_{TC})$;
10   if $(\beta, \theta) \neq no\_solution$
11     then
12       $ExecTest(\beta, \theta, no\_trace, M^{\top}_{TC})$;
13       terminate;
14   fi
15   $M^{\top}_{TC} := selectATC(M^{\top}_{CTG})$;
16 elihw
```

Fig. 4. Selection and execution of tests

In Figure 4, the test process is described as an algorithm. Its input parameters are a specification $Spec$ and a test purpose TP. $Spec$ is abstracted to $Spec^{\top}$ according to Section 5. Then $M^{\top}_{Spec}$ is generated from $Spec^{\top}$ and M_{TP} from TP. In parallel, a rule system $\mathcal{RS}$ is built, containing all conditions from $Spec$. $\mathcal{RS}$ will later be needed to parameterize test cases with concrete data. From the two $IOLTS$s, the complete test graph $M^{\top}_{CTG}$ is generated using TGV (cf. [3]). $M^{\top}_{CTG}$ may contain choices between several outputs to the IUT or even between inputs and outputs, so it is not necessarily controllable. Furthermore, $M^{\top}_{CTG}$ is an overapproximation of all test cases of the original system which satisfy the test purpose, so it may contain traces leading to unsound verdicts.

Our goal is to obtain parameterizable test cases (for instance in TTCN-3 [13]) together with information about data values to instantiate them. To make tests repeatable, we are interested in test cases where no nondeterministic choice is possible between several outputs or between inputs and outputs. Therefore, we single out a subgraph of $M^{\mathbb{T}}_{CTG}$ that contain neither choices between several outputs or choices between inputs and outputs, nor loops. We refer further to this subgraph as an abstract test case (ATC), denoted $M^{\mathbb{T}}_{TC}$.

Even though we are still working on the level of *IOLTS*s here, we now have to introduce variables for parameterization. In $M^{\mathbb{T}}_{TC}$, each occurrence of $\mathbb{T}$ is substituted by a unique symbolic variable v_{i_j} parameterizing inputs and outputs, respectively. The double index is necessary to identify the state, in which the transition with the variable starts (index i) and to uniquely identify this variable within the set of variables on transitions from state i (index j). These variables are embedded into the transition labels of the *IOLTS*, but are distinguished as a separate set *Var* in remainder of this section.

Definition 6.1 (Parameterizable Test Case) *Given a parameterized complete test graph* $M^{\mathbb{T}}_{CTG}(Var_{CTG}) = (\Sigma, Var_{CTG}, Lab, \rightarrow_{CTG}, \sigma_{init})$*, a* parameterized test case $M^{\mathbb{T}}_{TC}(Var_{TC})$ *is an input complete IOLTS* $(\Sigma, Var_{TC}, Lab, \rightarrow_{TC}, \sigma_{init})$ *such that* $Var_{TC} \subseteq Var_{CTG}$ *holds for the sets of symbolic variables of* $M^{\mathbb{T}}_{TC}$ *and* $M^{\mathbb{T}}_{CTG}$*. The set of states of the test case is a subset of the set of states of the complete test graph, and the test case shows only* Pass, Inconc *and* Fail *traces possible in the complete test graph, i.e.* $[\![M^{\mathbb{T}}_{TC}]\!]_{\mathsf{Pass}} \subseteq [\![M^{\mathbb{T}}_{CTG}]\!]_{\mathsf{Pass}}$, $[\![M^{\mathbb{T}}_{TC}]\!]_{\mathsf{Inconc}} \subseteq [\![M^{\mathbb{T}}_{CTG}]\!]_{\mathsf{Inconc}}$*, and* $[\![M^{\mathbb{T}}_{TC}]\!]_{\mathsf{Fail}} \subseteq [\![M^{\mathbb{T}}_{CTG}]\!]_{\mathsf{Fail}}$.

Before a parameterizable test case can be executed, it must be instantiated. This means, that each of the variables v_{i_j} must be set to a value such that a Pass-state in the test case can be reached with this valuation. In order to do so, a trace to Pass is selected with *NewPassTrace* (Figure 5). If such a trace exists, it can be executed, otherwise the next possible trace is searched. If no trace can be found in this abstract test case, the next test case is generated and examined for traces to Pass. If no such trace could be determined at all, the algorithm terminates with the final verdict None without executing any test cases.

The algorithm selects only one trace and executes it, where necessary dynamically adapting to the *IUT*'s reaction on input. A complete test suite consisting of more than one trace (irrespective of possible adaptions), could be executed by introducing a loop which repeats the trace selection and execution actions. The final verdict would then be the upper limit of verdicts for the single tests (see Definition 2.2).

Executing the trace β under a valuation θ does not mean, that this execution is bound to that trace for the whole execution. At some point during test execution, the *IUT* may nondeterministically leave the precalculated trace. In this case, the test execution algorithm tries to find another trace to a Pass verdict. This new trace, however, must contain the part of β, which has yet been executed, as its prefix. The trace valuation can also only be extended by new values.

$$\frac{l \longrightarrow_{g \,\triangleright\, !s(e)} \hat{l} \in Edg}{s(state(l, \overline{Var}), state(\hat{l}, \overline{Var}), param(e)) \coloneq g.}\ \text{ROutput}$$

$$\frac{l \longrightarrow_{?s(x)} \hat{l} \in Edg}{s(state(l, \overline{Var}), state(\hat{l}, \overline{Var}_{[x \mapsto Y]}), param(Y)).}\ \text{RInput}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, x:=e} \hat{l} \in Edg}{\tau(state(l, \overline{Var}), state(\hat{l}, \overline{Var}_{[x \mapsto e]}), param) \coloneq g.}\ \text{RAssign}$$

Table 4
Transformation of specification *Spec* into rule system $\mathcal{RS}$

Building the Rule System and Queries

A parameterizable test case may contain traces introduced by data abstraction. Moreover, information about the relationship of symbolic variables or concrete values they can be substituted with is absent. To sort out spurious traces and to obtain valuations for symbolic variables, we employ constraint-solving.

We transform the original specification *Spec* to a constraint logic program or a *rule system* $\mathcal{RS}$. This rule system forms the basis for test oracles or *queries*. A Pass-trace β which is selected from $M^{\mathbb{T}}_{TC}$ is transformed into a query $G := \mathcal{O}_\beta(\theta)$. Here, θ is a possible (i.e. valid) setting of data values, with which β can be instantiated. Let the set of symbolic variables in the specification be Var_{symb}. If there is no solution for the query, β is a spurious trace introduced by data abstraction and we remove the trace from the test case. If there is a solution $\theta : Var_{symb} \rightarrow D$ in $\mathcal{RS}$ for the query, β can be mapped to the trace of the original system.

We refer to trace β with symbolic variables substituted according to θ as an *instantiated trace* denoted $\beta(\theta)$. The instantiated trace $\beta(\theta)$ is a trace of the original system M_{Spec}. As we will prove later, the verdict assigned by $\beta(\theta)$ is sound. Knowing at least one possible solution for any Pass-trace is already enough to start executing test case $M^{\mathbb{T}}_{TC}$, parameterized with this solution θ. Further, we define the transformation of an original specification into a rule system and obtain a query from a Pass- or Inconc-trace of the test case.

Transformation from the original specification *Spec* to the rule system $\mathcal{RS}$ is defined by the inference rules given in Table 4. These rules map edges of the specification to rules of $\mathcal{RS}$. All the rules are of the form $rule\_name(state(l, \overline{Var}), state(\hat{l}, \overline{Var}'), param(Y)) \coloneq g$. The name of the rule is that of the corresponding action (τ for internal assignments). The first *state* parameter describes the source state of the edge in terms of the specification location and the process variables. The second *state* parameter describes the changed target state in the same terms. The third parameter *param* contains all symbolic variables or expressions which form the action parameters.

The rules RInput, ROutput and Assign transform input, output and internal assignment actions to constraint rules. The guard of an output or an internal action forms the body of the corresponding rule. Rules for input actions always holds, since no guards are specified for inputs. The action parameters *param* are given by the expression e for an output, or by a variable Y for an input action, while internal actions have no action parameters. Finally, input and internal assignment actions change values of variables. This change is noted in the second *state* parameter of the respective rule.

$$
\mathit{oracle}(i,r) =
\begin{cases}
[\,]\,, \textbf{ iff } r = [\,] \\
\big[\mathit{sig}(\tau)(\mathit{state}(l_{\mathit{init}}, \overline{\mathit{Var}}_{\mathit{init}}), \mathit{state}(l_1, \overline{\mathit{Var}}_1), \mathit{param}) | \mathit{oracle}(1, r')\big]\,, \\
\quad \textbf{iff } i = 0 \land r = [\sigma \rightarrow_\tau \sigma' | r'] \\
\big[\mathit{sig}(s)(\mathit{state}(l_{\mathit{init}}, \overline{\mathit{Var}}_{\mathit{init}}), \mathit{state}(l_1, \overline{\mathit{Var}}_1), \mathit{param}(Y)) | \mathit{oracle}(1, r')\big]\,, \\
\quad \textbf{iff } i = 0 \land \big(r = \big[\sigma \rightarrow_{?s(Y)} \sigma' | r'\big] \lor r = \big[\sigma \rightarrow_{!s(Y)} \sigma' | r'\big]\,\big) \\
\big[\mathit{sig}(\tau)(\mathit{state}(l_i, \overline{\mathit{Var}}_i), \mathit{state}(l_{i+1}, \overline{\mathit{Var}}_{i+1}), \mathit{param}) | \mathit{oracle}(i+1, r')\big]\,, \\
\quad \textbf{iff } i > 0 \land r = [\sigma \rightarrow_\tau \sigma' | r'] \\
\big[\mathit{sig}(s)(\mathit{state}(l_i, \overline{\mathit{Var}}_i), \mathit{state}(l_{i+1}, \overline{\mathit{Var}}_{i+1}), \mathit{param}(Y)) | \mathit{oracle}(i+1, r')\big]\,, \\
\quad \textbf{iff } i > 0 \land \big(r = \big[\sigma \rightarrow_{?s(Y)} \sigma' | r'\big] \lor r = \big[\sigma \rightarrow_{!s(Y)} \sigma' | r'\big]\,\big)
\end{cases}
$$

Table 5
Transformation of a trace of $M^{\mathbb{T}}_{TC}$ into oracle $\mathcal{O}_{TC}$

After the rule system $\mathcal{RS}$ has been generated, we proceed with choosing a Pass-trace β in $M^{\mathbb{T}}_{TC}$ and transforming it into an oracle $\mathcal{O}_\beta := \mathit{oracle}(0,\beta)$ using the function given in Table 5. Basically, an oracle is a sequence of rule invocations corresponding to the transitions along the chosen Pass-trace. Each transition along the trace is transformed into a rule invocation, which has the name of the action under consideration given as $\mathit{sig}(s)$. The parameters of this rule invocation are the state of the system where the transition starts (first parameter), the system's state after the transition and the action's parameters. In the first transition, which is characterized by the counter $i = 0$, the starting state of the transition is set to the initial state of the system. The function *oracle* then iterates through the trace and appends all rule invocations to one list, which forms the oracle.

In the oracle $\mathcal{O}_\beta$, all free variables in the system have not yet been bound to values. This happens by applying the constraint solver to the rule system $\mathcal{RS}$ and the oracle $\mathcal{O}_\beta$ using the function $\theta := \mathit{solve}(\mathcal{RS}, \mathcal{O}_\beta, \theta_{\mathit{const}})$.

Definition 6.2 (Partial Valuation) *Let vars* $: [\![M]\!]_{\mathit{trace}} \rightarrow \mathit{Var}_{\mathit{symb}}$ *be a function that projects the set of variables* $\mathit{Var}_{\mathit{symb}}$ *of* M *to that subset that is actually used in a given trace from* $[\![M]\!]_{\mathit{trace}}$.

Given a valuation $\theta : \mathit{vars}(\beta) \rightarrow D$ *and a trace* δ*, which is a prefix of* β*, we define the* partial valuation $\lfloor\theta\rfloor_\delta : \mathit{vars}(\delta) \rightarrow D$ *such that* $\forall x \in \mathit{vars}(\delta)\big(\lfloor\theta\rfloor_\delta(x) = \theta(x)\big)$.

The parameter $\theta_{const} \subseteq \theta$ can be used to define a set of constant valuation assignments. For instance, if a prefix δ of β has already been executed during a test and only for the suffix of β a new valuation has to be found, $\theta_{const} := \lfloor\theta\rfloor_\delta$ can be defined as this set of constant values. In all cases, where this situation is not applicable, i.e. no part of θ has to be constant, the optional parameter θ_{const} can be defined as $\emptyset$ and is further ignored. Having calculated a valuation θ for a trace β, the query $G := \mathcal{O}_\beta(\theta)$ can be built and it can be checked, whether $< G, true >$ is solvable.

When describing the test selection process in Section 6, the algorithm *NewPassTrace* has already been mentioned. Its task is to select a trace β from the abstract test case and find a valuation θ, so that $\beta(\theta)$ is a trace in the original system specification *Spec*. Therefore, the algorithm makes use of the oracle $\mathcal{O}_\beta$ and the rule system $\mathcal{RS}$. In the following lemma, we claim that if an oracle based on $\mathcal{RS}$ holds under a certain valuation then the corresponding trace under this valuation is a valid trace in M_{Spec} and vice versa.

Lemma 6.3 *Let $\beta(\theta)$ be a trace β of the ATC instantiated with the valuation θ. Then: $\mathcal{RS} \vdash \mathcal{O}_\beta(\theta) \Leftrightarrow \beta(\theta) \in [\![M_{Spec}]\!]_{trace}$.*

Proof sketch: To prove this lemma, both directions of implications have to be proven separately. For each of the directions, the initial and the general case of a transition in M_{Spec} are regarded separately for input, output and internal assignment actions. $\square$

Algorithm 6.2 (NewPassTrace($\delta, \theta, M_{TC}^{\top}$) : $(\beta, \theta') \in [\![M_{TC}^{\top}]\!]_{\mathsf{Pass}} \times \{Var_{symb} \to D\}$)

```
1   $\beta := selectFirst(\delta, [\![M_{TC}^{\top}]\!]_{\mathsf{Pass}})$;
2   while $\beta \neq no\_trace$
3     $\mathcal{O}_\beta := oracle(0, \beta)$
4     $\theta' := solve(\mathcal{RS}, \mathcal{O}_\beta, \lfloor\theta\rfloor_\delta)$
5     $G := \mathcal{O}_\beta(\theta')$
6     if $< G, true >$ is satisfiable
7       then return $(\beta, \theta')$;
8       else $\beta := selectNext(\delta, [\![M_{TC}^{\top}]\!]_{\mathsf{Pass}})$;
9     fi
10  elihw
11  return $no\_solution$;
```

Fig. 5. Pass trace selection procedure

The algorithm *NewPassTrace*, shown in Figure 5, finds a new trace to a Pass verdict together with a valid valuation. The algorithm takes a trace prefix δ, a valuation θ and a test case $M_{TC}^{\top}$ as input parameters and returns a trace $\beta \in [\![M_{TC}^{\top}]\!]_{\mathsf{Pass}}$ as well as an appropriate valuation (here θ'). It iterates over all possible

Algorithm 6.3 (ExecTest$(\beta, \theta, \delta, M^{\top}_{TC})$: $verdict \in \{\mathsf{None}, \mathsf{Pass}, \mathsf{Inconc}, \mathsf{Fail}\}$**)**

```
$step := next(\beta, \delta)$;
case $step$
  $no\_step$ :
    if $|\delta| > 0$
      then setVerdict(Pass);
      else setVerdict(None);
    fi
  $\tau$ : $ExecTest(\beta, \theta, add(\delta, step), M^{\top}_{TC})$;
  $!s(X)$: $sendToIUT(s([\![X]\!]_{\theta}))$;
    $ExecTest(\beta, \theta, add(\delta, step), M^{\top}_{TC})$;
  $?s(X)$: $receiveFromIUT(sig(Y))$;
    if $sig = s \wedge [\![Y]\!] = [\![X]\!]_{\theta}$;
      then $ExecTest(\beta, \theta, add(\delta, step), M^{\top}_{TC})$;
      else
        $\delta' := add(\delta, sig(Y))$;
        $\mathcal{O}_{\delta'} := oracle(0, \delta')$;
        $G_{\delta'} := \mathcal{O}_{\delta'}(\lfloor \theta \rfloor_{\delta[X \mapsto [\![Y]\!]]})$;
        if $\neg satisfiable(< G_{\delta'}, true >)$
          then setVerdict(Fail);
          else
            $(\beta', \theta') := NewPassTrace(\delta', \lfloor \theta \rfloor_{\delta[X \mapsto [\![Y]\!]]}, M^{\top}_{TC})$;
            if $(\beta', \theta') = no\_solution$
              then setVerdict(Inconc);
              else $ExecTest(\beta', \theta', \delta', M^{\top}_{TC})$;
            fi
        fi
    fi
esac
```

Fig. 6. Test execution procedure

traces to Pass with prefix δ in the test case and returns the first, which contains δ as its prefix and satisfies the query G under the valuation θ'. θ' is derived by solving the rule system $\mathcal{RS}$ for the trace β with a partial solution $\lfloor \theta \rfloor_{\beta}$ given. This partial solution cannot be changed anymore, since it gives the (proper) valuation for the already executed trace δ. The new trace found by *NewPassTrace* must satisfy $< \mathcal{O}_{\beta}(\theta'), true >$. If it does not, then the next possible trace to Pass is selected.

Test Execution

In an IUT, nondeterminism may be induced, for instance, by interleavings of the behavior of single components. In these cases, it is possible that during test execution the IUT leaves a trace to Pass which had been calculated beforehand and which is in principle a valid trace. Test execution then has to be adapted dynamically to the new situation.

Let β be a Pass-trace of $M^{\top}_{TC}$, θ be a solution for the query obtained from β by the rules in Table 5, and δ be the already executed prefix of β (initially it is empty). Let $next$ be a function that returns the next step of trace β or no_step, if no such step exists. Sending a signal to the IUT happens by the function $sendToIUT$, receiving by $receiveFromIUT$. Both functions are parameterized with the signal to be sent or received.

Test execution is defined by the recursive algorithm in Figure 6. First, the actual step under consideration is calculated. Then, a decision is made, based on the type of this step. If the next step is no_step, meaning that the end of the trace has been reached, the algorithm assigns either the None verdict, if no steps have yet been executed, or the Pass verdict. In this case, the test execution finished without finding any failures or inconclusive situations in the IUT. If the actual step is a τ-step, $ExecTest$ is invoked recursively, adding the τ-step to the trace prefix, which has already been executed before. An output step $!s(x)$ is treated similarly, except that the signal s is sent to the IUT. Its parameters are instantiated according to θ.

Handling an input $?s(X)$ is more complex. First, the input is received from the IUT as $?sig(Y)$. If now both the signal sig and the valuation of its parameters $[\![Y]\!]$ are as expected, then the step is just added to δ and a recursive invocation of the execution algorithm happens. If the signal sig or the parameter valuation does not fit the expectations, then it is checked, whether test execution has already left the valid traces of the system specification. In this case, Fail is assigned, otherwise a new trace to Pass with the new valuation is searched. If no such trace exists, Inconc is assigned. Otherwise, the algorithm is invoked recursively and test execution goes on.

Further, we argue the correctness of our approach by proving that the verdicts assigned to the IUT after having applied the algorithm $ExecTest$, are sound. Let $Spec$ be a specification and TP be a test purpose. Let $Spec^{\top}$ be a specification obtained by transforming $Spec$ by the rules in Table 2. Let $M^{\top}$ be an $IOLTS$ generated by the rules in Table 3 and $\mathcal{RS}_{Spec}$ be a rule system generated by the rules in Table 4.

First, the synchronous product $M^{\top}_{SP} \subseteq M^{\top}_{Spec} \times M_{TP}$ is built. From $M^{\top}_{SP}$, the abstract complete test graph $M^{\top}_{CTG}$ is derived. From $M^{\top}_{CTG}$, we get an abstract controllable test case $M^{\top}_{TC}$, from which we select a trace β to Pass. This trace is instantiated with data, which has been derived from a query to $\mathcal{RS}_{Spec}$. Trace β is then executed. Its already executed prefix is the trace δ.

Lemma 6.4 (Termination of Test Execution) *Given a finite (non-cyclic) trace β, the test execution algorithm (Figure 6) always terminates assigning a ver-*

dict, given that the IUT is deadlock-free.

Proof sketch: Given a finite trace, the algorithm always analyzes the next step in this trace. If there is no next step, i.e. the end of trace is reached, the algorithm terminates with Pass or None, resp. If the verdicts Fail or Inconc are assigned, the algorithm already stops without having reached the end of the trace under consideration. The lemma is proven by pointing out the exit points of the algorithm and the according assignments of verdicts. □

Lemma 6.5 (Soundness of verdicts) *The assignment of the test verdict to a test trace by the test execution algorithm (Figure 6) is sound.*

(i) *In case, that the verdict* Fail *is assigned, for* $\delta' = add(\delta, sig(Y))$ *holds:* $\delta'(\theta_{[X \mapsto [\![Y]\!]]}) \notin [\![M_{Spec}]\!]_{trace}$.

(ii) *In case, that the verdict* Inconc *is assigned, for the executed trace* δ *holds:* $\delta \in [\![M_{Spec}]\!]_{trace} \wedge \delta \notin [\![M_{TC}]\!]_{\mathsf{Pass}}$.

(iii) *If the verdict* Pass *is assigned, the executed trace* $\delta(\theta) \in [\![M_{Spec}]\!]_{trace} \wedge \delta \in [\![M^{\mathbb{T}}_{TC}]\!]_{\mathsf{Pass}} \wedge |\beta| > 0$.

(iv) *In case that* $|\beta| = 0$*,* None *is assigned.*

Proof sketch: This lemma is proven by first proving by induction, that any trace β which does not lead to a Fail verdict is a trace of the original system. The assignment of the separate verdicts is then shown by analyzing those points in the algorithm, which lead to the verdict under consideration, as well as the course of the algorithm from its initialization towards this point. □

7 CEPS Case Study

In this section, we describe the application of our approach to the case study *CEPS* (**C**ommon **E**lectronic **P**urse **S**pecifications). These define a protocol for electronic payment using a chip card as a wallet. The specifications consist of the *functional requirements* [6] and the *technical specification* [7]. A complete electronic purse system covers three roles: a card user, a card issuer (the issuing bank institute, for instance) and a card reader as a connection between these two. The hardware of such a system is given by the purse card itself, the card reader and some network infrastructure. Software applications are running on the card (`CEPCardApp`), on the card reader (`CERCardReaderApp`) and at the site of the card issuer (`CEPCardIssuerApp`), and these applications are communicating with each other.

In our work on the case study, we aim to evaluate our test generation process by automatically generating parameterizable test cases for the scenario described in Figure 7. We start from a formalized version of the technical specification of the CEPS card application `CEPCardApp` (Courtesy of the VASY team at INRIA Rhône-Alpes, cf. [15]), which we simplified by a live-variable-analysis and realized in μCRL. In this specification, all input variables with an infinite domain are substituted by $\mathbb{T}$. The generation process itself is guided by the *test purpose*, which describes the scenario, we are focusing on.

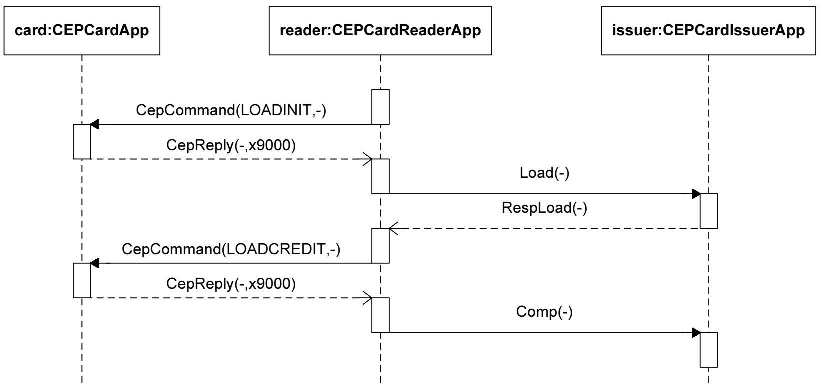

Fig. 7. Interactions in the test purpose scenario

The abstracted specification is then parsed and an LTS is generated. The resulting LTS is minimized using strong minimization. We experimented with two specifications for different scenarios. In the first one, a status variable of the process was after action `CepReply(updateStatus(mSlotInfo,x940A))` updated with value `x9409` instead of `x940A`. In the second specification, the value `x940A` was used.

For the first specification, the whole process of LTS generation and minimization took 16 minutes and 5.088 seconds on a cluster of five 2.2GHz AMD Athlon 64 bit single CPU computers with 1 GB RAM each. The abstracted specification had 3023122 states and 17459807 transitions, which could be reduced by strong minimization to 1627 states and 5487 transitions. Finally, two single test cases without loops are generated using TGV, one of them limited to a maximal depth search for its preamble of 100 steps, the other one unlimited. Starting with the minimized abstracted system model and a test purpose of 5 states and 5 transitions, the generated unlimited test case contained 594 states with 597 transitions. The limited test case contained 108 states with 111 transitions. Test case generation took 0.65 seconds or 0.42 seconds, resp., on a workstation with one 2.2GHz AMD Athlon XP 32 bit CPU and 1 GB main memory.

For the second specification, whose abstracted version had 168942 states and 232253 transitions (1619 states and 1899 transitions after strong minimization), the generation of the LTS took 69.418 seconds on the cluster. Test generation took 3.453 seconds for a test case of 255 states and 286 transitions (limitation to 100 steps led to identical results as unlimited generation).

Afterwards, a Prolog rule system is derived from the original specification that consists of the functions in the μCRL specification and of the conditions and assignments from the summands. This rule system is also reusable like the *IUT* model so that it potentially does not have to be regenerated each time a test oracle is created. The test oracle itself later delivers possible input and expected output values for the test execution. This oracle sends queries to the rule system to find out, under which variable settings the implemented trace can be executed and which values have to be expected from the *IUT*.

8 Conclusion and Future Work

In this paper, we proposed an approach to generate test cases combining data abstraction, enumerative test generation and constraint-solving. Given the concrete specification of an open system, the presented data abstraction allows to derive the appropriate abstract system that is finite with respect to data exchanged with its environment and thus suitable for the automatic generation of abstract test cases with enumerative tools. To execute the ATCs, we have to instantiate them with concrete data. For data selection we make use of constraint-solving techniques: a set of constraints is derived from the system specification and then solved by a constraint solver. The parameterized test cases can then be executed. We have proven the correctness of our approach. To corroborate the applicability of our approach, we applied it to the CEPS case study [5,6,7].

An interesting aspect, especially from a practical viewpoint, is the generation of test cases directly from UML specifications, as it has been proposed in [2,26]. As a future work, we aim to adapt our approach for UML-based test case generation. Doing so, the target language of test case generation is TTCN-3 (Testing and Test Control Notation, version 3), a standardized test specification language, widely accepted by the industrial community [35].

References

[1] Blom, S. C. C., W. J. Fokkink, J. F. Groote, I. A. van Langevelde, B. Lisser and J. C. van de Pol, *μCRL: A Toolset for Analysing Algebraic Specifications*, in: G. Berry, H. Comon and A. Finkel, editors, *Proc. of the 13th Intl. Conf. on Computer-Aided Verification*, LNCS **2102** (2001), pp. 250–254.

[2] Briand, L. C. and Y. Labiche, *A UML-Based Approach to System Testing*, in: *Proc. of the 4th Intl. Conf. on The Unified Modeling Language, Modeling Languages, Concepts, and Tools (UML'01)*, LNCS **2185** (2001), pp. 194–208.

[3] Calamé, J. R., *Specification-based Test Generation with TGV*, Technical Report SEN-R0508, Centrum voor Wiskunde en Informatica (2005).

[4] Calamé, J. R., N. Ioustinova and J. v. d. Pol, *Towards Automatic Generation of Parameterized Test Cases from Abstractions*, Technical Report SEN-E0602, Centrum voor Wiskunde en Informatica (2006), ISSN 1386-369X.

[5] Calamé, J. R., N. Ioustinova, J. v. d. Pol and N. Sidorova, *Data Abstraction and Constraint Solving for Conformance Testing*, in: *Proc. of the 12th Asia-Pacific Software Engineering Conf. (APSEC'05)* (2005), pp. 541–548.

[6] CEPSCO, "Common Electronic Purse Specifications, Functional Requirements," (1999), version 6.3.

[7] CEPSCO, "Common Electronic Purse Specifications, Technical Specification," (2000), version 2.2.

[8] Clarke, D., T. Jéron, V. Rusu and E. Zinovieva, *STG: A Symbolic Test Generation Tool*, in: *Proc. of the 8th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)* (2002), pp. 470–475.

[9] Clarke, L. A., *A System to Generate Test Data and Symbolically Execute Programs*, IEEE Transactions on Software Engineering **12** (1976), pp. 215–222.

[10] Cousot, P. and R. Cousot, *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, in: *Proc. of the 4th ACM SIGACT-SIGPLAN Symp. on Principles of programming languages (POPL'77)* (1977), pp. 238–252.

[11] Dams, D., "Abstract Interpretation and Partition Refinement for Model Checking," PhD dissertation, Eindhoven University of Technology (1996).

[12] Dams, D., R. Gerth and O. Grumberg, *Abstract Interpretation of Reactive Systems*, ACM Transactions on Programming Languages and Systems (TOPLAS) **19** (1997), pp. 253–291.

[13] ETSI ES 201 873-1 V2.2.1 (2003-02), *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*, ETSI Standard.

[14] Frantzen, L., J. Tretmans and T. Willemse, *Test Generation Based on Symbolic Specifications*, in: J. Grabowski and B. Nielsen, editors, *FATES 2004*, number 3395 in LNCS (2005), pp. 1–15.

[15] Garavel, H. and F. Lang, *NTIF: A General Symbolic Model for Communicating Sequential Processes with Data*, in: *Proc. of the 22nd IFIP WG 6.1 Intl. Conf. on Formal Techniques for Networked and Distributed Systems (FORTE'2002)*, 2002.

[16] Godefroid, P., N. Klarlund and K. Sen, *DART: Directed Automated Random Testing*, in: *Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'05)* (2005), pp. 213–223.

[17] Goodenough, J. B. and S. L. Gerhart, *Toward a Theory of Test Data Selection*, in: *Proc. of the Intl. Conf. on Reliable software* (1975), pp. 493–510.

[18] Groote, J. F. and A. Ponse, *The Syntax and Semantics of μCRL*, Technical report, Centrum voor Wiskunde en Informatica (1990).

[19] Groote, J. F., A. Ponse and Y. S. Usenko, *Linearization in parallel pCRL*, Journal of Logic and Algebraic Programming **48** (2001), pp. 39–72.

[20] Ioustinova, N., "Abstractions and Static Analysis for Verifying Reactive Systems," Ph.D. thesis, Free University of Amsterdam (2004).

[21] Ioustinova, N., N. Sidorova and M. Steffen, *Synchronous Closing and Flow Abstraction for Model Checking Timed Systems*, in: *Proc. of the 2nd Intl. Symp. on Formal Methods for Components and Objects (FMCO'03)*, LNCS **3188** (2004).

[22] Jard, C. and T. Jéron, *TGV: Theory, Principles and Algorithms*, Intl. Journal on Software Tools for Technology Transfer **7** (2005), pp. 297–315.

[23] Jeannet, B., T. Jéron, V. Rusu and E. Zinovieva, *Symbolic Test Selection based on Approximate Analysis*, in: *Proc. of the 11th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, number 3440 in LNCS, Edinburgh (Scottland), 2005.

[24] King, J. C., *Symbolic Execution and Program Testing*, Commun. of the ACM **19** (1976), pp. 385–394.

[25] Marriott, K. and P. J. Stuckey, "Programming with Constraints – An Introduction," MIT Press, Cambridge, 1998.

[26] Offutt, J. and A. Abdurazik, *Generating tests from uml specifications*, in: *Proc. of the 2nd Intl. Conf. on The Unified Modeling Language, Modeling Languages, Concepts, and Tools (UML'99)*, 1999, pp. 416–429.

[27] OMG, "UML 2.0 Testing Profile Specification," (2003), version 2.0, Final Adopted Specification/finalization phase.

[28] Pretschner, A., O. Slotosch, E. Aiglstorfer and S. Kriebel, *Model-based Testing for Real*, Intl. Journal on Software Tools for Technology Transfer **5** (2004), pp. 140–157.

[29] Rusu, V., L. d. Bousquet and T. Jeron, *An Approach to Symbolic Test Generation*, in: *Proc. of the Intl. Conf. on Integrating Formal Methods (IFM'00)*, LNCS **1945**, 2000, pp. 338–357.

[30] Sidorova, N. and M. Steffen, *Embedding Chaos*, in: P. Cousot, editor, *Proc. of the 8th Intl. Static Analysis Symp.*, LNCS **2126** (2001), pp. 319–334.

[31] Tillmann, N. and W. Schulte, *Parameterized Unit Tests*, in: *Proc. of the 10th European Software Engineering Conf. (ESEC/FSE-13), held jointly with 13th ACM SIGSOFT Intl. Symp. on Foundations of Software Engineering* (2005), pp. 253–262.

[32] Tretmans, J., *Test Generation with Inputs, Outputs, and Repetitive Quiescence*, Software - Concepts & Tools **17** (1996), pp. 103–120.

[33] Tretmans, J. and E. Brinksma, *TorX: Automated Model-based Testing*, in: A. Hartman and K. Dussa-Ziegler, editors, *1st European Conf. on Model-Driven Software Engineering*, 2003.

[34] Visser, W., C. S. Pasareanu and S. Khurshid, *Test Input Generation with Java PathFinder*, in: *Proc. of the 2004 ACM SIGSOFT Intl. Symp. on Software Testing and Analysis(ISSTA 2004)* (2004), pp. 97–107.

[35] Willcock, C., T. Deiß, S. Tobies, S. Keil, F. Engler and S. Schulz, "An Introduction to TTCN-3," Wiley, 2005.

[36] Zinovieva-Leroux, E., "Méthodes symboliques pour la génération de tests de systèmes reactifs comportant des données," Ph.D. thesis, Université de Rennes (2004).