Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

Towards automatic generation of parameterized test cases from abstractions

J.R. Calamé, N. Ioustinova, J.C. van de Pol

REPORT SEN-E0602 MARCH 2006

Centrum voor Wiskunde en Informatica (CWI) is the national research institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organisation for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Towards automatic generation of parameterized test cases from abstractions

ABSTRACT
Model-based tools for automatic test generation usually can handle systems of a rather limited size. Therefore, they cannot be applied directly to systems of real industrial size. Here, we propose an approach to test generation combining enumerative data abstraction, test generation methods and constraint solving. The approach allows applying enumerative test generation tools like TGV to large and infinite systems. Given such a system, abstractions allow to derive a finite abstract system suitable for automatic test generation with enumerative tools. Abstract test cases need to be parameterized with actual test data, in order to execute them. For data selection, we make use of constraint solving techniques. Test case execution will later be done by TTCN-3.

1998 ACM Computing Classification System: D.2.1, D.2.5
Keywords and Phrases: conformance testing; test case generation; data abstraction; constraint-solving
Note: This work was carried out under the ITEA project TT-Medal

Towards Automatic Generation of Parameterized Test Cases from Abstractions

Jens R. Calamé, Natalia Ioustinova, Jaco van de Pol
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
Model-based tools for automatic test generation usually can handle systems of a rather limited size. Therefore, they cannot be applied directly to systems of real industrial size. Here, we propose an approach to test generation combining enumerative data abstraction, test generation methods and constraint solving. The approach allows applying enumerative test generation tools like TGV to large and infinite systems. Given such a system, abstractions allow to derive a finite abstract system suitable for automatic test generation with enumerative tools. Abstract test cases need to be parameterized with actual test data, in order to execute them. For data selection, we make use of constraint solving techniques. Test case execution will later be done by TTCN-3.

1998 ACM Computing Classification System: D.2.1 [Requirements/Specification], D.2.5 [Testing and Debugging].
Keywords and Phrases: conformance testing, test case generation, data abstraction, constraint-solving.
Note: This work was carried out under the ITEA project TT-Medal.

1. Introduction

The test of a software product is a crucial aspect in every software developing process. Therefore, various test approaches have been developed over the last decades. Depending on the position of test case development in the software development process, they can roughly be categorized into *test-first* and *test-last* approaches. In test-first approaches like unit testing in an extreme programming process [1] (XP) require the software developer to write test cases before the code of the implementation under test (IUT). In many cases, these test cases are then at the same time the specification of the IUT. However, the effects of the test-first approach are not without controversy. In only half the studies presented in [13], a positive effect on the software quality could be determined, which in only one case was definitely not accompanied by a decrease in productivity.

In this paper, we concentrate on test-last approaches, like they are custom in more formal development processes, e.g. the Rational Unified Process [27]. Here, the software product is modeled and specified before its implementation and test. To support modeling software over different phases of a development process using models of different granularity and degree of abstraction, the Model-Driven Architecture (MDA [29]) has been developed by the OMG. Separating software specification and implementation allows to apply this approach to the development of the *IUT* and its test cases. The MDA divides the documents produced during software development into *platform-independent models* (PIM), *platform-specific models* (PSM) and *code*. This has the advantage, that software can be evolved or ported more easily than with a specification, which is already platform-specific code like in XP. The disadvantage of course is, that test cases are not created "automatically". Automatic test generation tries to eliminate this shortcoming.

Conformance testing [36] is one of the most rigorous among existing testing techniques, checking whether an IUT is consistent with its specification. This is the case only if every observable behavior of the IUT is allowed by the specification. Regarding the MDA and testing, we are working on the most abstract view on a system, the PIM, here. From this PIM we generate platform-independent test models and finally platform-independent test code. The generated test cases do not reject consistent

IUTs and do not accept IUTs showing behaviors not allowed by the specification. One major problem of automatic test generation is its termination or the number of generated test cases, resp. If the generation process is not guided except for the specification documents, it may not terminate at all or produce many unnecessary test cases around a few useful ones. Thus, we have to make an attempt to focus the generation on certain aspects, like the main risks of an IUT (*risk-based testing*). This aim can be reached by introducing *test purposes*, which describe the relevant behavioral aspects for a set of test cases.

Not only the selection of behavior is crucial for successful test generation, but also that of data. Software is in most cases interacting with an environment, which stimulates the execution of interface actions parameterized with data values coming from large or even infinite domains. Considering these parameters already at the stage of test generation leads to problems similar to the ones described above. Thus, we have to abstract away from concrete data and concentrate on behavioral aspects only for test case generation, and reintroduce data for test execution.

In this paper, we present such a test generation framework. Starting from a PIM of an IUT and an appropriate test purpose, we abstract away input and output data from the IUT. The abstract system then shows at least the behavior of the original system [20]. Afterwards, *abstract* test cases are generated, which contain a control flow and are parameterizable with concrete data values during test execution. These data values can be obtained from a rule system, which is set up in parallel and serves data intervals for data selection and as a test oracle.

We implement our approach to generate test cases from μCRL specifications with TGV. TGV (Test Generation with Verification technology) [22] is a tool for the automatic generation of test cases from formal specifications of reactive systems. TGV implements algorithms based on adaptation of on-the-fly model-checking algorithms. Test selection in TGV is based on the concept of test purposes. However, specifications of systems operating on large or infinite data domains are beyond the scope of TGV even with on-the-fly test generation using test purposes. μCRL (micro Common Representation Language) is a specification language based on process algebra with data. We use this language to specify systems and test purposes. A μCRL toolset supports state space generation, reduction techniques, optimization techniques based on static analysis and abstractions. Prolog is used to implement constraint solving for data selection and the test oracle.

This paper is organized as follows. In Section 2, we give an introduction to the theory of conformance testing, which our approach is based on. Section 3 defines syntax and semantics of the systems, we are working with. Section 4 discusses the necessary preliminaries of constraint solving. The approach of data abstraction is introduced in Section 5. Section 6 discusses the determination of test case parameters. In Section 7, we work out the application of our approach to the CEPS [6] case study before we conclude in Section 8, also presenting an overview of related work.

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 [34], 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, which is described in [22]. We do not yet discuss the problem of quiescence here.

Definition 1 *((Deterministic) IOLTS).* An input output labeled transition system *IOLTS* is a tuple $M = (\Sigma, Lab, \rightarrow, \sigma_{init})$ where

- $\Sigma \neq \emptyset$ is a set of states,
- Lab is a set of labels (*actions*),
- $\rightarrow \subseteq \Sigma \times Lab \times \Sigma$ is a transition relation,
- $\sigma_{init} \in \Sigma$ is the initial state.

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 only one outgoing transition for each action $\lambda \in Lab$ in each state $\sigma \in \Sigma$. ■

The behavior of an *IOLTS* is given by sequences of states and transitions $\beta = \sigma_{init} \rightarrow \sigma_1 \rightarrow \ldots$ starting from the initial state. In traces, the states are projected out, i.e. $[\![M]\!]_{trace} \subseteq Lab^\star$. We use $[\![M]\!]_{trace}$ to denote the set of traces of an *IOLTS* M.

Definition 2 *(Trace).* A *trace* β of M is a mapping $\beta_\lambda : N \longrightarrow Lab$, where either $N = \{0, 1, 2, \ldots, n\}$ or $N = \mathbb{N}$, and there exists a mapping $\beta_\sigma : N \longrightarrow \Sigma$ such that $(\beta_\sigma(i) \rightarrow_{\beta_\lambda(i+1)} \beta_\sigma(i+1)) \in \rightarrow$ for all $i, (i+1) \in N$. If $N = \mathbb{N}$, trace β is called an *infinite* trace; otherwise, it is called a *finite* trace. The *length* of β is defined as $|N|$ and referred to as $|\beta|$.

In the further text, we refer to the set of traces in M as $[\![M]\!]_{trace}$. ■

Definition 3 *(Relation **after**).* The relation **after** is defined for states and action labels as $\Sigma \times Lab \rightarrow \mathcal{P}(\Sigma)$ with $\Sigma' = \sigma$ **after** λ being the set of states which can be reached from σ by a transition labeled with λ.

For traces and action labels, the relation **after** is defined as $[\![M]\!]_{trace} \times Lab \rightarrow \mathcal{P}(\Sigma)$ with $\Sigma' = \beta$ **after** λ being the set of states which can be reached by a transition labeled with λ after trace β. ■

*IOLTS*s modeling *IUT*s are assumed to be *input-complete*, meaning, 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 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[1].

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. Moreover, M_{TP} is complete in all the states except of the accepting and refusing ones.

Definition 4 *(Trap State).* In an *IOLTS* $M = (\Sigma, Lab, \rightarrow, \sigma_{init})$, a *trap state* is a state $\sigma \in \Sigma$ for which $\text{trap} : \Sigma \rightarrow \{\text{true, false}\}$ defined as $\text{trap}(\sigma) = \forall(\sigma, \lambda, \sigma') \in \rightarrow (\sigma = \sigma')$ holds. ■

Definition 5 *((Complete) Test Purpose).* Let the *IOLTS* $M_{Spec} = (\Sigma^{Spec}, Lab^{Spec}, \rightarrow^{Spec}, \sigma_{init}^{Spec})$ be a specification. A test purpose is a (complete) *deterministic IOLTS* $M_{TP} = (\Sigma^{TP}, Lab^{TP}, \rightarrow^{TP}, \sigma_{init}^{TP})$ with a set of labels $Lab^{TP} = Lab_I^{Spec} \cup Lab_O^{Spec} \cup \{ACCEPT, REFUSE\}$ (internal actions of *Spec* are not considered here).

Let furthermore be:

- $\Sigma_{acc}^{TP} = \{\sigma | \sigma \in \Sigma^{TP} \wedge \text{trap}(\sigma) \wedge \exists t \in \rightarrow^{TP} \big(t = (\sigma, ACCEPT, \sigma)\big)\} \subseteq \Sigma^{TP}$ and
- $\Sigma_{ref}^{TP} = \{\sigma | \sigma \in \Sigma^{TP} \wedge \text{trap}(\sigma) \wedge \exists t \in \rightarrow^{TP} \big(t = (\sigma, REFUSE, \sigma)\big)\} \subseteq \Sigma^{TP}$.

Σ_{acc}^{TP} is the set of *accept states* of the test purpose, Σ_{ref}^{TP} the set of *refuse states*. The following must hold for a test purpose:

$$\Sigma_{acc}^{TP} \neq \emptyset \wedge \Sigma_{acc}^{TP} \cap \Sigma_{ref}^{TP} = \emptyset.$$

Transitions labeled with *ACCEPT* or *REFUSE* are allowed in trap states only. ■

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. 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 only matching the action *names* of M^{Spec} and M^{TP}.

[1] The difference with *ioco* [34] is that we do not abstract from τ-steps and that we do not yet consider quiescence.

Definition 6 *(Synchronous Product of System Specification and Test Purpose [22]).* The synchronous product M_{SP} of the system specification $M_{Spec} = (\Sigma^{Spec}, Lab^{Spec}, \rightarrow^{Spec}, \sigma_{init}^{Spec})$ and a complete test purpose $M_{TP} = (\Sigma^{TP}, Lab^{TP}, \rightarrow^{TP}, \sigma_{init}^{TP})$ is the *IOLTS* $M_{SP} = M_{Spec} \times M_{TP} = (\Sigma^{SP}, Lab^{SP}, \rightarrow^{SP}, \sigma_{init}^{SP})$ where:

- $Lab^{SP} = Lab^{Spec} \cup \{ACCEPT, REFUSE\}$ is the alphabet of the *IOLTS*,
- $\rightarrow^{SP}$ is the set of transitions such that $((\sigma, \sigma''), \lambda, (\sigma', \sigma''')) \in \rightarrow^{SP} \Leftrightarrow \Big(\big((\sigma, \lambda, \sigma') \in \rightarrow^{Spec} \wedge (\sigma'', \lambda, \sigma''') \in \rightarrow^{TP} \big) \vee \big((\sigma'', ACCEPT, \sigma''') \in \rightarrow^{TP} \wedge \sigma = \sigma' \big) \vee \big((\sigma'', REFUSE, \sigma''') \in \rightarrow^{TP} \wedge \sigma = \sigma' \big) \Big)$, and
- $\sigma_{init}^{SP} = (\sigma_{init}^{Spec}, \sigma_{init}^{TP}) \in \Sigma^{SP}$ is the initial state.

■

Definition 7 *(Complete Test Graph [22]).* The complete test graph *CTG* is an *IOLTS* $M_{CTG} = (\Sigma^{CTG}, Lab^{CTG}, \rightarrow^{CTG}, \sigma_{init}^{CTG})$ which is determined from the synchronous product M_{SP} in the following way:

1. The set of actions is determined by mirroring the set of actions of M_{CTG}: $Lab^{CTG} = Lab_I^{CTG} \cup Lab_O^{CTG}$ with
   - $Lab_O^{CTG} \subseteq Lab_I^{Spec}$,
   - $Lab_I^{CTG} = Lab_O^{Spec}$.

   The reason for mirroring inputs and outputs lies in the relation between a test case and the implementation under test, 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 SUT, its set of outputs Lab_O^{CTG} is limited to a subset of the SUT's set of inputs Lab_I^{Spec}.
2. The set of states is determined. This set is divided into four subsets $\Sigma^{CTG} = \underbrace{\Sigma_{L2A}^{CTG}}_{\supseteq \Sigma_{\mathsf{Pass}}^{CTG}} \dot\cup \Sigma_{\mathsf{Inconc}}^{CTG} \dot\cup \Sigma_{\mathsf{Fail}}^{CTG}$

   which are defined as follows:

   Lead to Accept: $\Sigma_{L2A}^{CTG} = \{\sigma \in \Sigma^{SP} | \exists \beta \in [\![M_{SP}]\!]_{trace} (\sigma \rightarrow_\beta \sigma' \wedge \sigma' \in \Sigma_{acc}^{SP})\}$,
   Pass: The set $\Sigma_{\mathsf{Pass}}^{CTG} \subseteq \Sigma_{L2A}^{CTG}$ is defined as $\Sigma_{\mathsf{Pass}}^{CTG} = \Sigma_{acc}^{SP}$. This set may not be empty.
   Inconclusive: $\Sigma_{\mathsf{Inconc}}^{CTG} = \{\sigma' | \exists \sigma \in \Sigma_{L2A}^{CTG}, \sigma' \notin \Sigma_{L2A}^{CTG}, \iota \in Lab_O^{SP} (\sigma \rightarrow_\iota \sigma' \in \rightarrow^{SP})\}$,
   Fail: $\Sigma_{\mathsf{Fail}}^{CTG} = \{\sigma_{\mathsf{Fail}}^{CTG}\}$, $\sigma_{\mathsf{Fail}}^{CTG} \notin \Sigma^{SP}$.

   For reasons of manageability of the resulting IOLTS, the state $\sigma_{\mathsf{Fail}}^{CTG}$ exists only implicitly and is assumed as end point for all possible traces $\sigma \notin [\![M_{SP}]\!]_{trace}$. It is not actually generated.
3. The set of transitions of the *CTG* is defined as $\rightarrow^{CTG} = \rightarrow_{L2A}^{CTG} \cup \rightarrow_{\mathsf{Inconc}}^{CTG} \cup \rightarrow_{\mathsf{Fail}}^{CTG}$ with:
   - $\rightarrow_{L2A}^{CTG} = \rightarrow^{SP} \cap (\Sigma_{L2A}^{CTG} \times Lab^{CTG} \times \Sigma_{L2A}^{CTG})$,
   - $\rightarrow_{\mathsf{Inconc}}^{CTG} = \rightarrow^{SP} \cap (\Sigma_{L2A}^{CTG} \times Lab_I^{CTG} \times \Sigma_{\mathsf{Inconc}}^{CTG})$,
   - $\rightarrow_{\mathsf{Fail}}^{CTG} = \{\sigma \rightarrow_{\mathsf{Fail}}^{CTG} | \sigma \in \Sigma_{L2A}^{CTG} \wedge \lambda \in Lab_I^{CTG} \wedge \sigma \textbf{ after } \lambda = \emptyset\}$.

The *CTG* may contain loops and choices between several outputs in the same state or between inputs and outputs and is thus *not (necessarily) controllable.* ■

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, where $[\![M_{SP}]\!]_{atrace} \subseteq [\![M_{SSpec}]\!]_{trace}$ and $[\![M_{SP}]\!]_{rtrace} = [\![M_{SSpec}]\!]_{trace} \backslash [\![M_{SP}]\!]_{atrace}$. Depending on the trace, executed during the actual test, a verdict is assigned.

Definition 8 *(Verdict).* A *verdict* is the result of the execution of a test case. It is determined by the comparison between the actual behavior of the IUT during test case execution and its expected behavior. In general, there exist five types of verdicts of which we consider the following four: Pass, Inconc, Fail and None.

The verdict is set by a function setverdict : $[\![M_{SP}]\!]_{trace} \rightarrow$ Verdict which is defined as follows:

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

The types of verdicts can be structured in the following partial order: None $\sqsubseteq$ Pass $\sqsubseteq$ Inconc $\sqsubseteq$ Fail (cf. [3], Definition 30). ■

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_{SSpec} but the trace does not satisfy the test purpose (traces from $[\![M_{SP}]\!]_{rtrace}$). The Fail verdict is implicit. All unspecified outputs lead to this verdict. During generation of the M_{CTG}, all input and output actions are mirrored so that the set of input actions of the M_{CTG} equals the set of output actions of the M_{SSpec} and the set of output actions of the M_{CTG} is a subset of the set of input actions of the M_{SSpec}.

As we said before, M_{CTG} may contain choices between several outputs and choices between inputs and outputs. Controllable test cases are derived by resolving these choices, meaning, a test case does not contain these choices between outputs or between inputs and outputs anymore.

Definition 9 *((Controllable) Test Case).* A test case is a deterministic input complete *IOLTS* $M_{TC} = (\Sigma^{TC}, Lab^{TC}, \rightarrow^{TC}, \sigma^{TC}_{init})$ derived from M_{CTG} with

- $\Sigma^{TC} \subseteq \Sigma^{CTG}$: $\Sigma^{TC} = \Sigma^{TC}_{L2A} \cup \Sigma^{TC}_{\mathsf{Pass}} \cup \Sigma^{TC}_{\mathsf{Inconc}} \cup \Sigma^{TC}_{\mathsf{Fail}}$ and $\Sigma^{TC}_{L2A} \subseteq \Sigma^{CTG}_{L2A}$, $\Sigma^{TC}_{\mathsf{Pass}} \subseteq \Sigma^{CTG}_{\mathsf{Pass}}$, $\Sigma^{TC}_{\mathsf{Inconc}} \subseteq \Sigma^{CTG}_{\mathsf{Inconc}}$, $\Sigma^{TC}_{\mathsf{Fail}} \subseteq \Sigma^{CTG}_{\mathsf{Fail}}$;
- $Lab^{TC} \subseteq Lab^{CTG}$: $Lab^{TC}_{O} \subseteq Lab^{CTG}_{O} \wedge Lab^{TC}_{I} = Lab^{CTG}_{I}$;
- $\rightarrow^{TC} \subseteq \rightarrow^{CTG}$: $\rightarrow^{TC} = \rightarrow^{TC}_{L2A} \cup \rightarrow^{TC}_{\mathsf{Inconc}} \cup \rightarrow^{TC}_{\mathsf{Fail}}$ and $\rightarrow^{TC}_{L2A} \subseteq \rightarrow^{CTG}_{L2A}$, $\rightarrow^{TC}_{\mathsf{Inconc}} \subseteq \rightarrow^{CTG}_{\mathsf{Inconc}}$, $\rightarrow^{TC}_{\mathsf{Fail}} \subseteq \rightarrow^{CTG}_{\mathsf{Fail}}$;
- $\sigma^{TC}_{init} = \sigma^{CTG}_{init}$.

Similarly to the sets of accepted and refused traces in CTG, the final states of a test case induce the sets of traces leading to Pass (Inconc or Fail) verdict denoted $[\![M_{TC}]\!]_{\mathsf{Pass}}$ ($[\![M_{TC}]\!]_{\mathsf{Inconc}}$ or $[\![M_{TC}]\!]_{\mathsf{Fail}}$, resp.). ■

The test cases, we treat in this paper, are loopfree and controllable. They are executed in parallel with an IUT. The traces in a test case are chosen in a way that one trace leads to a Pass state. From this trace, several branches lead to Inconc states in one step. These Inconc states represent traces in the test purpose which end in a refusing state.

Definition 10 *(Soundness [31]).* Given a specification M_{SSpec}, a test purpose M_{TP} and a test case M_{TC}, a *verdict* of a test case M_{TC} is *sound* iff the following holds for the executed trace β:

sort $Bool$
func $T :\rightarrow Bool \qquad F :\rightarrow Bool$
map $and : Bool \times Bool \rightarrow Bool$
var b: $Bool$
rew $and(T, b) = b \qquad and(b, T) = b \qquad and(F, F) = F$

Figure 1: Data type for booleans

$$\text{setverdict}(\beta) = \begin{cases} \mathsf{Pass} & \Leftrightarrow \beta \in [\![M_{TP}]\!]_{atrace} \cap [\![M_{Spec}]\!]_{trace} \\ \mathsf{Inconc} & \Leftrightarrow \beta \in [\![M_{TP}]\!]_{rtrace} \cap [\![M_{Spec}]\!]_{trace} \\ \mathsf{Fail} & \Leftrightarrow \beta \notin [\![M_{Spec}]\!]_{trace} \end{cases}$$

The verdict None is not considered here, since it is a construct to assign a test verdict to a trace, which has not yet been executed.

If for all traces in a test case, a sound verdict is assigned, the test case is sound. ■

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 easily be too large or even infinite due to all possible data.

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 its signature $Sig(Spec) = (Sort, Fun, Act, Comm, Proc)$ [18]. It specifies an open system that communicates with its environment. *Sort* defines a set of data types for the declaration of variables. Each S consists of a set of constructors, which have the form $c :\rightarrow S$ or $c : S_1 \times \ldots \times S_n \rightarrow S$, resp., with $S_1, \ldots, S_n \in Sort$. These constructors are used to form typed values D_S of sort S ($D_S \rightarrow S$).

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 1 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* is 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* [2]. To make the explanations in this paper easier to understand for the reader, however, we give a definition of processes based on the theory of Symbolic Transition Systems (STSs). 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[2]. A mapping of variables to values is called a valuation; we denote the set of valuations by $Val = \{\eta \mid \eta\colon Var \rightarrow D\}$. 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 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 containing a value to be assigned to a local variable, (2) *output* of a signal s together with a value described by an expression, and (3) *assignments*. Every

[2]In an LPO, location would just be one additional parameter in the set of process parameters for a single process, or n such additional parameters for n parallel processes.

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

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

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

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

action except inputs is *guarded* by a boolean expression g, its guard. The three classes of actions are written as $?s(x)$, $g \rhd !s(e)$, and $g \rhd x := e$, 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 $\zeta = \sigma_{init} \rightarrow \sigma_1 \rightarrow \ldots$ starting from the initial one. The step semantics is given by an *IOLTS* $M = (\Sigma, Lab, \rightarrow, \sigma_{init})$, where $\rightarrow \subseteq \Sigma \times Lab \times \Sigma$ is given as 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, $l \longrightarrow_{g \rhd !s(e)} \hat{l} \in Edg$, 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, $l \longrightarrow_{g \rhd x:=e} \hat{l} \in Edg$, result in the change of location and the 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. For the linear process operators [2], the realization of our approach is working on, existing linearization techniques [17] allow to obtain a single process definition for a parallel composition of a finite number of process definitions by resolving communication and parallel composition.

Assumption 1 *(Treatment of Data in Test Purposes).* In a test purpose M_{TP}, we assume that the information about an action carried in the labels of Lab^{TP} is limited to the names of the actions. Data parameters should not eb subject of value assignment and are thus replaced by the *don't-care* parameter $*$. ■

4. Constraint Solving Preliminaries

In this section we give an overview of notions related to constraint solving [28].

A *constraint domain* $\mathcal{D}$ consists of a set of n-ary constraint symbols which describe relations, a logical theory $\mathcal{T}$ and for every constraint symbol c a tuple of value sets $\langle V_1, \ldots, V_n \rangle$. An example for such a constraint symbol is "$\leq$". A *primitive constraint* $c(X_1, \ldots, X_n)$ 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 $C = c_1 \wedge \ldots \wedge c_m$ where $m \geq 0$ and $c_1, \ldots, c_m$ 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 $\langle V_1, \ldots, V_n \rangle$ 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 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 one or more solutions.

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* is returned in all those cases, in which a solution might exist, but could not be determined by $solv(C)$.

A *user defined* constraint is of the form $p(t_1, \ldots, t_n)$ where p is an n-ary predicate and $t_1, \ldots, t_n$ are terms (variables, constants or functions) from a constraint domain. An example for a user-defined constraint is $a(X, f(Y)) : -X > 0, Y < X$, which takes two parameters (the variable X and the function $f(Y)$, with Y again being a variable) and incorporates the two primitive constraints $X > 0$ and $Y < X$. A *literal* is either a primitive constraint or a user defined constraint. A *rule* R is of the form $A :\!-\ B$ where A is a user defined constraint and B is a sequence of literals. A *fact* is a rule with an empty sequence of literals, i.e. a rule of the form $A :\!-\ \square$, where $\square$ 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. $G = L_1, \ldots, L_m$ with $m > 0$. If $m = 0$, then G is an *empty* query denoted $\square$. Let query G be of the form $L_1, \ldots, L_{(i-1)}, L_i, L_{(i+1)}, \ldots, L_m$ and $L_i = p(s_1, \ldots, s_n)$. Let rule R be of form $p(s_1, \ldots, s_n) :\!-\ B$. The *renaming* of G at L_i by R using ϱ then is the goal $L_1, \ldots, L_{(i-1)}, s_1 = \varrho(t_1), \ldots, s_n = \varrho(t_n), \varrho(B), L_{(i+1)}, \ldots, L_m$, where ϱ is chosen so that variables from $\varrho(R)$ do not appear in G.

Definition 11 *(Derivation).* Let G be a query and $\mathcal{P}$ be a CLP program. A *state* of the program is a pair $\langle G \mid C \rangle$ where C is a constraint storage.

Let the program be in state $\langle G_i \mid C_i \rangle$, where $G_i = L_1, \ldots, L_n$. A *derivation step* from $\langle G_i \mid C_i \rangle$ to $\langle G_{(i+1)} \mid C_{(i+1)} \rangle$ denoted $\langle G_i \mid C_i \rangle \Rightarrow \langle G_{(i+1)} \mid C_{(i+1)} \rangle$, is defined as follows:

If L_1 is a primitive constraint, then $C_{(i+1)} = C_i \wedge L_1$. If $C_{(i+1)}$ is not satisfiable then $G_{(i+1)}$ is an empty goal. Otherwise, $G_{(i+1)} = L_2, \ldots, L_n$.

If L_1 is a user-defined constraint, then $C_{(i+1)} = C_i$ and $G_{(i+1)} = \varrho(L_1), L_2, \ldots, L_n$, where $\varrho(L_1)$ is the renaming of the constraint L_1 in G_i to guarantee that no variable names used in earlier steps occur.

A *derivation* is a sequence of derivation steps $\langle G_1 \mid C_1 \rangle \Rightarrow \langle G_2 \mid C_2 \rangle \Rightarrow \ldots \Rightarrow \langle G_n \mid C_n \rangle$. A derivation for a query G starts with the initial state $\langle G \mid \mathit{true} \rangle$. A derivation is *successful* if it ends in a state $\langle \square \mid C \rangle$ with C for which solver $solv(C)$ returns either *true* or *unknown*.

The constraint storage C of the final state $\langle \square \mid C \rangle$ is named an *answer* to the goal G.

A valuation θ is a *solution* for query G if $\mathcal{D} \models [\![C]\!]_\theta$ where C is an answer to $\langle G \mid \mathit{true} \rangle$. ■

5. Data Abstraction

Signals coming from the chaotic environment can carry *any* value. This often boosts the state space of the system to infinity. We do not make any assumptions about values from environment, meaning one can conceptually abstract values influenced by the environment via inputs and assignments to one abstract *"chaotic"* value, denoted $\mathbb{T}$. That basically means ignoring these values and focusing on the control structure of a process. 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 [33] 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 behaviors of the original system [33, 21].

sort	$Bool^{\top}$
func	$\top_{Bool}: \rightarrow Bool^{\top}$
	$\kappa_{Bool}: Bool \rightarrow Bool^{\top}$
map	$and^{\top}: Bool^{\top} \times Bool^{\top} \rightarrow Bool^{\top}$
	$\gamma: Bool^{\top} \rightarrow Bool$
var	$b, b': Bool$
rew	$and^{\top}(\kappa(b), \kappa(b')) = \kappa(and(b, b'))$
	$and^{\top}(\top_{Bool}, \kappa(F)) = and^{\top}(\kappa(F), \top_{Bool}) = \kappa(F)$
	$and^{\top}(\top_{Bool}, \kappa(T)) = and^{\top}(\kappa(T), \top_{Bool}) = \top_{Bool}$
	$and^{\top}(\top_{Bool}, \top Bool) = \top_{Bool}$
	$\gamma(\top_{Bool}) = T$
	$\gamma(\kappa(b)) = b$

Figure 2: Transformed sort $Bool^{\top}$

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, $\top_S :\rightarrow S^{\top}$ and $\kappa : S \rightarrow S^{\top}$. 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 $m : S_1 \times \cdots \times S_n \rightarrow S_{n+1}$, we define a mapping $m^{\top} : S_1^{\top} \times \cdots \times S_n^{\top} \rightarrow S_{n+1}^{\top}$ 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 $m^{\top}$:

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

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 an occurrence of the 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 expression $e^{\top}$ of sort $S^{\top}$. Thereby, all the newly introduced symbols (constructors and rewrite rules) are used and replace the appropriate 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 guards evaluating to *true* or *false*, respectively. The guards valuated to $\top$ behave as guards evaluating to *true*. We implement this by introducing an extra mapping $\gamma: Bool^{\top} \rightarrow Bool$ that is *true* whenever a guard is evaluated either to $\top$ or to $\kappa(true)$ and *false* otherwise. To avoid introducing unnecessary nondeterminism, we apply a more refined transformation to the sort $Bool$. Its abstraction, sort $Bool^{\top}$, is illustrated in Fig. 2.

Definition 12 *(May Semantics for Chaotic Guards).* While a guard g is defined as a function $g : D \rightarrow \{\text{true, false}\}$, a chaotic guard is defined as a function $g^{\top} : D^{\top} \rightarrow \{\text{true, false, } \top\}$. To map this three value logic back to a two value logic, a *may*-function $\gamma : \{\text{true, false, } \top\} \rightarrow \{\text{true, false}\}$ is defined as follows: $\gamma(\kappa(true)) = true$, $\gamma(\kappa(false)) = false$ and $\gamma(\top) = true$. ■

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 $\top$-value of the proper sort followed by assigning this $\top$-value to

the variable x (see rule $\textsc{Input}^{\top}$ in Table 2). Assignments and outputs are treated w.r.t. the rules in Table 1. The semantics of the transformed system are given by the inference rules in Table 3.

$$\begin{array}{rcl} \mathit{Spec} & \xrightarrow{\mathit{Tab.}\ 2} & \mathit{Spec}^{\top} \\ \mathit{Tab.}\ 1 \big\downarrow & & \big\downarrow \mathit{Tab.}\ 3 \\ M & \preceq_{\leq} & M^{\top} \end{array}$$

$M^{\top}$ can receive only $\top$ 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 $\top$ values. This means, that $M^{\top}$ simulates M. Further, we give an overview of preservation results based on [21, 19].

Definition 13 *(≤-Simulation).* Let $M_1 = (\Sigma^1, \mathit{Lab}^1, \rightarrow^1, \sigma_0^1)$ and $M_2 = (\Sigma^2, \mathit{Lab}^2, \rightarrow^2, \sigma_0^2)$ be two *IOLTS*s. $(\leq_a, \leq_b)$ is a simulation, iff $\forall \sigma_1, \hat{\sigma}_1, \sigma_2, \lambda_1 \exists \hat{\sigma}_2, \lambda_2 \big(\sigma_1 \leq_a \sigma_2 \wedge \sigma_1 \rightarrow_{\lambda_1} \hat{\sigma}_1 \Rightarrow (\lambda_1 \leq_b \lambda_2 \wedge \hat{\sigma}_1 \leq_a \hat{\sigma}_2 \wedge \sigma_2 \rightarrow_{\lambda_2} \hat{\sigma}_2)\big)$, $\sigma_1, \hat{\sigma}_1 \in \Sigma^1, \lambda_1 \in \mathit{Lab}^1, \sigma_2, \hat{\sigma}_2 \in \Sigma^2, \lambda_2 \in \mathit{Lab}^2$.

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 $\sigma^1_{init} \leq_a \sigma^2_{init}$. ■

This simulation relation is now defined for concrete and abstracted *IOLTS*s. 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 $\mathit{Loc} \times \mathit{Val}$ of the original system with the states of the transformed system $\mathit{Loc} \times \mathit{Val}^{\top}$, we define the relation $\leq_S$ on states as $\leq_S : \Sigma \times \Sigma^{\top}$.

Definition 14 *(Relation $\leq_S$).* Let $\sigma = (l, \eta)$ and $\sigma^{\top} = (l, \eta^{\top})$ be two states of the *IOLTS*s M and $M^{\top}$ with specifications *Spec* and $\mathit{Spec}^{\top}$. $\leq_S : \Sigma \times \Sigma^{\top}$ is defined as $\sigma \leq_S \sigma^{\top}$ iff $\forall x \in \mathit{Var}\big([\![x]\!]_{\eta^{\top}} = \top \vee [\![x]\!]_{\eta^{\top}} = \kappa([\![x]\!]_{\eta})\big)$. ■

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

Definition 15 *(Relation $\leq_L$).* Let $\lambda \in \mathit{Lab}$ and $\lambda^{\top} \in \mathit{Lab}^{\top}$. Then $\lambda \leq_L \lambda^{\top}$ is defined as follows:

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

■

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

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

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

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

Table 2: Transformation of edges ($\mathit{Spec} \rightarrow \mathit{Spec}^{\top}$)

Proof. The lemma can easily be proven based on Definition 13. Let a specification *Spec* and its abstracted counterpart $Spec^{\top}$ be given as follows:

$$\begin{aligned} Spec &= \big(\mathcal{Z}, \mathcal{E}, (Var, Loc, \sigma_{init}, Edg)\big) \text{ with } \sigma_{init} = (l_{init}, \eta_{init}) \\ Spec^{\top} &= \big(\mathcal{Z}^{\top}, \mathcal{E}^{\top}, (Var, Loc, \sigma^{\top}_{init}, Edg^{\top})\big) \text{ with } \sigma^{\top}_{init} = (l_{init}, \eta^{\top}_{init}) \end{aligned}$$

η_{init} is the initial valuation of all global variables in *Spec*. Analogously, $\eta^{\top}_{init}$ is the initial valuation of all global variables in $Spec^{\top}$ with some values possibly being set to $\top$. The semantics of the specifications are given by two *IOLTS*s M and $M^{\top}$ defined as follows:

$$\begin{aligned} M &= (\Sigma, Lab, \rightarrow, \sigma_{init}) \\ M^{\top} &= (\Sigma^{\top}, Lab^{\top}, \rightarrow^{\top}, \sigma^{\top}_{init}) \end{aligned}$$

Furthermore, let $\sigma, \hat{\sigma} \in \Sigma$, $\sigma \rightarrow_{\lambda} \hat{\sigma}$ in the set of transitions and $\sigma^{\top} \in \Sigma^{\top}$ be given. We prove the lemma by first considering the relationship between the initial states of both systems and then considering the relation of an arbitrary step.

Initial step – $\sigma_{init} \leq_S \sigma^{\top}_{init}$: We consider the initial states first. $\sigma^{\top}_{init} = (l_{init}, \eta^{\top}_{init})$ is derived from $\sigma_{init} = (l_{init}, \eta_{init})$ by substituting either none or some or all variable values in η_{init} by $\top$, while the control location l_{init} stays the same all three cases. Substituting values in this way leads to $\eta^{\top}_{init}$. The relation $\leq_S$ holds here. If $\eta^{\top}$ is initialized with the original values from M, $\forall x \in Var\big([\![x]\!]_{\eta^{\top}_{init}} = \kappa([\![x]\!]_{\eta_{init}})\big)$ holds. If all initialization values are set to $\top$, $\forall x \in Var([\![x]\!]_{\eta_{init}} = \top)$ holds. For a mixed valuation, *Var* can be divided into exactly two disjunct subsets for which the two conditions above hold as well so that Definition 14 is fully satisfied.

General step – $\lambda \leq_L \lambda^{\top} \wedge \sigma \leq_S \sigma^{\top}$: Now, we consider the general step. Assume that $\sigma \leq_S \sigma^{\top}$ and let $\sigma \rightarrow_{\lambda} \hat{\sigma}$ in M be given. Under these conditions, we can prove that:

- $\exists \sigma^{\top} \rightarrow_{\lambda^{\top}} \hat{\sigma}^{\top}$ in the set of transitions of $M^{\top}$ such that $\lambda \leq_L \lambda^{\top}$ and
- $\hat{\sigma} \leq_S \hat{\sigma}^{\top}$.

$$\begin{array}{ccc} \sigma & \leq_S & \sigma^{\top} \\ \lambda \big\downarrow & \leq_L & \big\downarrow \lambda^{\top} \\ \hat{\sigma} & \leq_S & \hat{\sigma}^{\top} \end{array}$$

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

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

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

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

We have to prove that an appropriate transition $\sigma^{\mathbb{T}} \rightarrow_{\lambda^{\mathbb{T}}} \hat{\sigma^{\mathbb{T}}}$ is generated in $M^{\mathbb{T}}$ with $\hat{\sigma} \leq_S \hat{\sigma}^{\mathbb{T}}$. In order to do so, we have to distinguish three different cases, namely input actions, output actions and τ-steps. In all three cases, the action in M starts in a state $\sigma = (l, \eta)$, that in $M^{\mathbb{T}}$ starts in a state $\sigma^{\mathbb{T}} = (l, \eta^{\mathbb{T}})$ with $\sigma \leq_S \sigma^{\mathbb{T}} \Leftrightarrow (l, \eta) \leq_S (l, \eta^{\mathbb{T}})$.

Input action: Let λ be $?s(x)$. The semantics of the step $l \rightarrow_{?s(x)} \hat{l}$ is given for M in table 1 as $(l, \eta) \rightarrow_{?s(v)} (\hat{l}, \eta_{[x \mapsto v]})$. For $M^{\mathbb{T}}$, rule INPUT$^{\mathbb{T}}$, given in table 2, transforms the step to $l \rightarrow_{?s(\mathbb{T})} \rightarrow_{true \,\triangleright\, x:=\mathbb{T}} \hat{l}$. Its step semantics, given as rule INPUT$^{\mathbb{T}}$ in table 3, finally leads to $(l, \eta^{\mathbb{T}}) \rightarrow_{?s(\mathbb{T})} (\hat{l}, \eta^{\mathbb{T}}_{[x \mapsto \mathbb{T}]})$. It holds that $?s(v) \leq_L ?s(\mathbb{T})$, since $?s(v) \leq_L ?s(v')$ with $v' = \mathbb{T}$. Furthermore, $(\hat{l}, \eta_{[x \mapsto v]}) \leq_S (\hat{l}, \eta^{\mathbb{T}}_{[x \mapsto \mathbb{T}]})$, since $\forall x \in Var\Big(\big([\![x]\!]_{\eta^{\mathbb{T}}} = \kappa([\![x]\!]_{\eta})\big) \vee \big([\![x]\!]_{\eta^{\mathbb{T}}} = \mathbb{T}\big)\Big)$. With $\hat{\sigma} = (\hat{l}, \eta_{[x \mapsto v]})$ and $\hat{\sigma}^{\mathbb{T}} = (\hat{l}, \eta^{\mathbb{T}}_{[x \mapsto \mathbb{T}]})$, this immediately leads to $\hat{\sigma} \leq_S \hat{\sigma}^{\mathbb{T}}$.

Output action: Let λ be $g \triangleright !s(e)$. The step semantics for $l \rightarrow_{g \,\triangleright\, !s(e)} \hat{l}$ given in table 1 is $(l, \eta) \rightarrow_{!s(v)} (l, \eta)$ with $[\![e]\!]_{\eta} = v$ for M and $(l, \eta^{\mathbb{T}}) \rightarrow_{\gamma(g^{\mathbb{T}}) \,\triangleright\, !s(e^{\mathbb{T}})} (l, \eta^{\mathbb{T}})$ for $M^{\mathbb{T}}$. The fact, that $\eta \leq \eta^{\mathbb{T}}$ holds has already been shown above. It is guaranteed that this step appears in $M^{\mathbb{T}}$, since $\forall g(g \Rightarrow \gamma(g^{\mathbb{T}}))$ per definition (Definition 12). As defined in Definition 15, $!s(v) \leq_L !s(v^{\mathbb{T}})$ holds for $v^{\mathbb{T}} = \kappa(v)$. If $v^{\mathbb{T}}$ is influenced by $\mathbb{T}$ in $M^{\mathbb{T}}$, this leads to $!s(v) \leq_L !s(\mathbb{T})$, which also holds as of Definition 15. $\hat{\sigma} \leq_S \hat{\sigma}^{\mathbb{T}}$ for the same reason as shown above; there has even been no change in the valuation η or $\eta^{\mathbb{T}}$, resp.

τ-step: Let λ be an assignment $x := e$. The semantics of the step $l \rightarrow_{g \,\triangleright\, x:=e} \hat{l}$ is given for M in table 1 as $(l, \eta) \rightarrow_{\tau} (\hat{l}, \eta_{[x \mapsto e]})$. For $M^{\mathbb{T}}$, rule INPUT$^{\mathbb{T}}$, given in table 2, transforms the step to $l \rightarrow_{\gamma(g) \,\triangleright\, x:=e^{\mathbb{T}}} \hat{l}$. Its step semantics, given as rule INPUT$^{\mathbb{T}}$ in table 3, finally leads to $(l, \eta^{\mathbb{T}}) \rightarrow_{\tau} (\hat{l}, \eta^{\mathbb{T}}_{[x \mapsto e^{\mathbb{T}}]})$. It is trivial to show that $\tau \leq_L \tau$ holds, since we have equal actions here without any data parameters. Furthermore, $(\hat{l}, \eta_{[x \mapsto e]}) \leq_S (\hat{l}, \eta^{\mathbb{T}}_{[x \mapsto e^{\mathbb{T}}]})$, since $\forall x \in Var\Big(\big([\![x]\!]_{\eta^{\mathbb{T}}} = \kappa([\![x]\!]_{\eta})\big) \vee \big([\![x]\!]_{\eta^{\mathbb{T}}} = \mathbb{T}\big)\Big)$. With $\hat{\sigma} = (\hat{l}, \eta_{[x \mapsto e]})$ and $\hat{\sigma}^{\mathbb{T}} = (\hat{l}, \eta^{\mathbb{T}}_{[x \mapsto e^{\mathbb{T}}]})$, this immediately leads to $\hat{\sigma} \leq_S \hat{\sigma}^{\mathbb{T}}$.

□

Using Lemma 1, it is easy to show that every trace of M can be mimicked by a trace of $M^{\mathbb{T}}$.

Definition 16. Let (σ^M, σ^N) be a state of $M \times N$ and (σ^O, σ^P) be a state of $O \times P$. Then the following holds: $(\sigma^M, \sigma^N) \leq_S (\sigma^O, \sigma^P) \Leftrightarrow \sigma^M \leq_S \sigma^O \wedge \sigma^N \leq_S \sigma^P$. ■

Lemma 2. For the synchonous product of automata S and T with P holds: $\forall S, T, P(S \preceq_{\leq} T \Rightarrow S \times P \preceq_{\leq} T \times P)$ under the simulation relation $(\leq_S, \leq_L)$. ■

Proof. Let $\sigma^S, \hat{\sigma}^S \in \Sigma^S$, $\sigma^T \in \Sigma^T$ and $\sigma^P \in \Sigma^P$ be arbitrary states in the automata's state spaces. We assume that $(\sigma^S, \sigma^P) \leq_S (\sigma^T, \sigma^P)$, since $S \preceq_{\leq} T$ following Definition 16. Furthermore, we can assume (see proof for Lemma 1) that $\hat{\sigma}^S \leq_S \hat{\sigma}^T$, $\hat{\sigma}^T \in \Sigma^T$. As of Definition 16, $(\hat{\sigma}^S, \hat{\sigma}^P) \leq_S (\hat{\sigma}^T, \hat{\sigma}^P)$ since both $\hat{\sigma}^S \leq_S \hat{\sigma}^T$ and $\hat{\sigma}^P \leq_S \hat{\sigma}^P$ (trivial).

$$\begin{array}{rcl} (\sigma^S, \sigma^P) & \leq_S & (\sigma^T, \sigma^P) \\ \lambda \big\downarrow & \leq_L & \big\downarrow \lambda^{\mathbb{T}} \\ (\hat{\sigma}^S, \hat{\sigma}^P) & \leq_S & (\hat{\sigma}^T, \hat{\sigma}^P) \end{array}$$

Now let there be $\sigma^S \rightarrow^S_{\lambda} \hat{\sigma}^S$, $\sigma^T \rightarrow^T_{\lambda} \hat{\sigma}^T$ and $\sigma^P \rightarrow^P_{\lambda} \hat{\sigma}^P$. Building the synchronous product over the automata leads to the transitions $(\sigma^S, \sigma^P) \rightarrow^{S \times P}_{\lambda} (\hat{\sigma}^S, \hat{\sigma}^P)$ and $(\sigma^T, \sigma^P) \rightarrow^{T \times P}_{\lambda} (\hat{\sigma}^T, \hat{\sigma}^P)$. Building the synchronous product over two of the automata does not change anything about the action λ under consideration, so if $\lambda \leq_L \lambda$ holds for S and T, it will do the same for $S \times P$ and $T \times P$. □

Definition 17 *(≤-inclusion on traces).* Let ζ and ρ be traces of *IOLTS*s 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 18 *(≤-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 3. 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. Let $M \times M_{TP}$ be the synchronous product of a system and a test purpose and let $M^\top \times M_{TP}$ be the synchronous product of an abstracted system and the same test purpose. For both products, the sets of traces are defined as the disjunct set of accepting traces, which end in an acceptance state, and refusing traces ending in a refusal state. That means $[\![M \times M_{TP}]\!]_{trace} = [\![M \times M_{TP}]\!]_{atrace} \dot\cup [\![M \times M_{TP}]\!]_{rtrace}$ and $[\![M^\top \times M_{TP}]\!]_{trace} = [\![M^\top \times M_{TP}]\!]_{atrace} \dot\cup [\![M^\top \times M_{TP}]\!]_{rtrace}$. We have to show that the following holds:

1. $[\![M \times M_{TP}]\!]_{atrace} \subseteq_\leq [\![M^\top \times M_{TP}]\!]_{atrace}$ and
2. $[\![M \times M_{TP}]\!]_{rtrace} \subseteq_\leq [\![M^\top \times M_{TP}]\!]_{rtrace}$.

As we have proved before, $M \times M_{TP} \preceq_\leq M^\top \times M_{TP}$ holds (lemmata 1 and 2). It follows from definitions 17 and 18 that $[\![M \times M_{TP}]\!]_{atrace} \dot\cup [\![M \times M_{TP}]\!]_{rtrace} \subseteq_\leq [\![M^\top \times M_{TP}]\!]_{atrace} \dot\cup [\![M^\top \times M_{TP}]\!]_{rtrace}$. Herefrom, we can conclude that

1. $[\![M \times M_{TP}]\!]_{atrace} \subseteq_\leq [\![M^\top \times M_{TP}]\!]_{atrace} \dot\cup [\![M^\top \times M_{TP}]\!]_{rtrace}$ and
2. $[\![M \times M_{TP}]\!]_{rtrace} \subseteq_\leq [\![M^\top \times M_{TP}]\!]_{atrace} \dot\cup [\![M^\top \times M_{TP}]\!]_{rtrace}$.

To prove the above claim, we have to show that

1. $[\![M \times M_{TP}]\!]_{atrace} \cap [\![M^\top \times M_{TP}]\!]_{rtrace} = \emptyset$ and
2. $[\![M \times M_{TP}]\!]_{rtrace} \cap [\![M^\top \times M_{TP}]\!]_{rtrace} = \emptyset$.

This is easy to show. There exists no trace $\rho \in [\![M \times M_{TP}]\!]_{atrace}$ which would ever end in a refusal state, so we can conclude immediately $\rho \in [\![M \times M_{TP}]\!]_{atrace} \Rightarrow \rho \notin [\![M^\top \times M_{TP}]\!]_{rtrace}$. Analogously, we can say that $\rho \in [\![M \times M_{TP}]\!]_{rtrace} \Rightarrow \rho \notin [\![M^\top \times M_{TP}]\!]_{atrace}$. For this reason, our entrance claim holds. □

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 over the whole process. Then, we describe how the necessary rule system is built and how test selection and execution work algorithmically. Finally, we review our approach and prove the soundness of verdicts derived from test execution.

Test Process Overview

In Figure 3, the test process is described as an algorithm (for its graphical representation cf. Figure 10). Its input parameters are a specification *Spec* and a test purpose *TP*. *Spec* is abstracted to $Spec^\top$ according to Section 5. Then $M_{Spec}^\top$ 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 $M_{CTG}^\top$ is generated using TGV (cf. [3]). $M_{CTG}^\top$ may contain choices between several outputs to the *IUT* or even between inputs and outputs, so it is not necessarily controllable. Furthermore, $M_{CTG}^\top$ is an overapproximation of all

Algorithm 1 *(SelectAndExecTest(Spec, TP) : verdict* $\in$ {None, Pass, Inconc, Fail}*).*

```
1  setVerdict(None);
2  Spec^⊤ := abstract(Spec);
3  RS := buildRuleSystem(Spec);
4  M^⊤_Spec := generateLTS(Spec^⊤);
5  M_TP := generateLTS(TP);
6  M^⊤_CTG := generateCTG(M^⊤_Spec, M_TP);
7  M^⊤_TC := selectATC(M^⊤_CTG);
8  while M^⊤_TC ≠ no_testcase
9    (β, θ) := NewPassTrace(no_trace, ∅, M^⊤_TC);
10   if (β, θ) ≠ no_solution
11     then
12       ExecTest(β, θ, no_trace, M^⊤_TC);
13       terminate;
14   fi
15   M^⊤_TC := selectATC(M^⊤_CTG);
16 elihw
```

Figure 3: Selection and execution of tests

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 [14]) together with information about values that can be used to instantiate them. Hence, 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 $M^{\top}_{CTG}$ 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 $M^{\top}_{TC}$.

Even though we are still working on the level of *IOLTS*s here, we now have to introduce variables for parameterization. In $M^{\top}_{TC}$, each occurrence of $\top$ 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 the further regard.

Definition 19 *(Parameterizable Test Case).* Given a complete test graph $M^{\top}_{CTG} = (K, Var_{CTG}, Lab, \rightarrow_{CTG}, k_{init})$, a *parameterized test case* $M^{\top}_{TC}$ is an input complete *IOLTS* $(K', Var_{TC}, Lab, \rightarrow_{TC}, k_{init})$ where the set Var_{TC} of symbolic variables of $M^{\top}_{TC}$ is a subset of the set Var_{CTG} of symbolic variables of $M^{\top}_{CTG}$; 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. $[\![M^{\top}_{TC}]\!]_{\mathsf{Pass}} \subseteq [\![M^{\top}_{CTG}]\!]_{\mathsf{Pass}}$, $[\![M^{\top}_{TC}]\!]_{\mathsf{Inconc}} \subseteq [\![M^{\top}_{CTG}]\!]_{\mathsf{Inconc}}$, and $[\![M^{\top}_{TC}]\!]_{\mathsf{Fail}} \subseteq [\![M^{\top}_{CTG}]\!]_{\mathsf{Fail}}$. ■

Before such a test case can be executed, it must be instantiated. This means, that each of the variables v_{i_j} must be preset with 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*. This algorithm is described in more detail in Section 6. If such a trace exists, it can be executed under the given valuation. If this is not the case, then the next possible trace is searched. If no such trace can be found in the abstract test case under consideration, the next test case is generated and examined for traces to Pass. If no such trace could be determined at all, then the algorithm terminates with the final verdict None, since it has not executed any test cases. Please remark, that only one trace is

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

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

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

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

selected and executed by this algorithm, even though there might be more. A complete test suite consisting of more than one traces, 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 8).

Executing the trace β does not mean, that the test execution algorithm is bound to the trace for the whole execution. At some point during test execution, the IUT may leave the precalculated trace. In case, that the SUT nondeterministically decides to leave β, 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. This is also the case for the valuation of this trace prefix. The valuation can also only be extended by new values for the alternative trace.

Building the Rule System and Queries
A parameterized 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 information about 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 can then be queried. Each Pass-trace β which is selected from $M_{TC}^{\top}$ is transformed into a query $G := \mathcal{O}_\beta(\theta)$. 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 $\theta : Var_{symb} \to D$ in $\mathcal{RS}$ for the query, the trace 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. 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 $M_{TC}^{\top}$. 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 $rule\_name(state(l, \overline{Var}), state(\hat{l}, \overline{Var}'), param(Y)) :\!-\ g$. 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 are local for this edge. These are the action parameters.

Rule ROutput transforms an output edge $l \longrightarrow_{g \rhd !s(e)} \hat{l}$ into a rule $s(state(l, \overline{Var}), state(\hat{l}, \overline{Var}), param(e)) :\!-\ g$. The name of the rule coincides with the signal s. The edge leads to change of location from l to $\hat{l}$. The values of the process variables $\overline{Var}$ remain unmodified. The signal is parameterized

with a value given by expression e that becomes a parameter of the *param*-part of the rule. The rule holds only if the guard is satisfied.

Rule RInput transforms an input edge $l \longrightarrow_{?s(x)} \hat{l}$ into a rule $s(state(l, \overline{Var}),\ state(\hat{l}, \overline{Var}_{[x \mapsto Y]}),\ param(Y))$. Here, input leads to the substitution of process variable x by a symbolic variable Y that is local for this rule.

Rule RAssign maps an assign-edge $l \longrightarrow_{g \rhd x:=e} \hat{l}$ into a τ-rule $\tau(state(l, \overline{Var}),\ state(\hat{l}, \overline{Var}_{[x \mapsto e]}),\ param) :\!-\ g$. The rule is satisfied only if the guard g is satisfied. An assignment is represented by substituting process variable x by expression e. τ-rules have no local parameters, thus the structure *param* has the arity 0.

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

Table 5: Transformation of a trace of $M_{TC}^{\top}$ into oracle $\mathcal{O}_{TC}$

After the rule system $\mathcal{RS}$ has been generated, we proceed with choosing a Pass-trace β in $M_{TC}^{\top}$ and transforming it into an oracle $\mathcal{O}_\beta := oracle(0, \beta)$ using the function given in Table 5. Basically, a 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 $sig(s)$. 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 $\theta := solve(\mathcal{RS}, \mathcal{O}_\beta, \theta_{const})$.

Definition 20 *(Partial Valuation).* Let $vars : [\![M]\!]_{trace} \rightarrow Var_{symb}$ be a function that projects the set of variables Var_{symb} of M to that subset that is actually used in a given trace from $[\![M]\!]_{trace}$.

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

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 $\mathcal{RS}$.

Lemma 4. 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. To show the equivalence stated above, we have to divide the proof into two parts, proving each direction separately.

$\mathcal{RS} \vdash \mathcal{O}_\beta(\theta) \Leftarrow \beta(\theta) \in [\![M_{Spec}]\!]_{trace}$ We begin with proving the lemma for test oracles consisting of only one step.

1. Let $\beta(\theta)$ be a trace of *one* transition $\sigma_{init} \rightarrow_\tau \hat{\sigma} \equiv (l_{init}, \overline{Var}_{init}) \rightarrow_\tau (\hat{l}, \overline{Var}_{init[x \mapsto v]})$. As defined in the step semantics in Table 1 (rule Assign), the appropriate step in the specification is $l_{init} \longrightarrow_{g \rhd x:=e} \hat{l}$ with $[\![g]\!]_\theta = \mathit{true}$ and $v = [\![e]\!]_\theta$. According to rule RAssign in Table 4, the rule system $\mathcal{RS}$ contains the rule $\tau(\mathit{state}(l_{init}, \overline{Var}_{init}), \mathit{state}(\hat{l}, \overline{Var}_{init[x \mapsto e]}), \mathit{param}) :- g$. Following the function *oracle* in Table 5, the oracle $\mathcal{O}_\beta$ only contains the rule invocation $\tau(\mathit{state}(l_{init}, \overline{Var}_{init}), \mathit{state}(l_1, \overline{Var}_1), \mathit{param})$. This oracle holds since
   (a) $\mathcal{RS}$ contains the appropriate rule (see above),
   (b) this rule instantiates e with $[\![e]\!]_\theta$ when invoked, and
   (c) this rule holds for $\hat{l} = l_1$, $\overline{Var}_{init[x \mapsto e]} = \overline{Var}_1$ under valuation θ and $[\![g]\!]_\theta = \mathit{true}$.
2. Let $\beta(\theta)$ be a trace of *one* transition $\sigma_{init} \rightarrow_{!s(v)} \hat{\sigma} \equiv (l_{init}, \overline{Var}_{init}) \rightarrow_{!s(v)} (\hat{l}, \overline{Var}_{init})$. As defined in the step semantics in Table 1 (rule Output), the appropriate step in the specification is $l \rightarrow_{g \rhd !s(e)} \hat{l}$ with $[\![g]\!]_\theta = \mathit{true}$ and $v = [\![e]\!]_\theta$. According to rule ROutput in Table 4, the rule system $\mathcal{RS}$ contains the rule $s(\mathit{state}(l_{init}, \overline{Var}_{init}), \mathit{state}(\hat{l}, \overline{Var}_{init}), \mathit{param}(e)) \coloneq g$. Following the function *oracle* in Table 5, the oracle $\mathcal{O}_\beta$ only contains the rule invocation $s(\mathit{state}(l_{init}, \overline{Var}_{init}), \mathit{state}(l_1, \overline{Var}_1), \mathit{param}(v))$ with $v = [\![e]\!]_\theta$. This oracle holds since
   (a) $\mathcal{RS}$ contains the appropriate rule (see above),
   (b) this rule instantiates e with $[\![e]\!]_\theta$ when invoked, and
   (c) this rule holds for $\hat{l} = l_1$, $\overline{Var}_{init} = \overline{Var}_1$ under valuation θ and $[\![g]\!]_\theta = \mathit{true}$.
3. Let $\beta(\theta)$ be a trace of *one* transition $\sigma_{init} \rightarrow_{?s(v)} \hat{\sigma} \equiv (l_{init}, \overline{Var}_{init}) \rightarrow_{?s(v)} (\hat{l}, \overline{Var}_{init[x \mapsto v]})$. As defined in the step semantics in Table 1 (rule Input), the appropriate step in the specification is $l_{init} \rightarrow_{?s(x)} \hat{l}$. According to rule RInput in Table 4, the rule system $\mathcal{RS}$ contains the rule $s(\mathit{state}(l_{init}, \overline{Var}_{init}), \mathit{state}(\hat{l}, \overline{Var}_{init[x \mapsto Y]}), \mathit{param}(Y))$. Following the function *oracle* in Table 5, the oracle $\mathcal{O}_\beta$ only contains the rule invocation $s(\mathit{state}(l_{init}, \overline{Var}_{init}), \mathit{state}(l_1, \overline{Var}_1), \mathit{param}(v))$. Thus, this oracle holds since
   (a) $\mathcal{RS}$ contains the appropriate rule (see above),
   (b) this rule instantiates Y with v when invoked, and
   (c) this rule holds for $\hat{l} = l_1$ and $\overline{Var}_{[x \mapsto Y]} = \overline{Var}_1$ under valuation θ.

Since we assume the constraint-solver to work correctly, this will be the case if $\beta(\theta) \in [\![M_{Spec}]\!]_{trace}$.

Now we regard the general step of the proof. We assume to have an oracle $\mathcal{O}_\delta$, which holds under θ since $\delta \in [\![M_{Spec}]\!]_{trace}$. We extend δ by one transition to completely describe trace $\beta \in [\![M_{Spec}]\!]_{trace}$ with the prefix δ.

4. Let $\beta(\theta)$ be a trace with the prefix δ followed by the transition $\sigma \rightarrow_\tau \hat{\sigma} \equiv (l, \overline{Var}) \rightarrow_\tau (\hat{l}, \overline{Var}_{[x \mapsto v]})$. As defined in the step semantics in Table 1 (rule Assign), the appropriate step in the specification is $l \longrightarrow_{g \rhd x:=e} \hat{l}$ with $[\![g]\!]_\theta = \mathit{true}$ and $v = [\![e]\!]_\theta$. According to rule RAssign in

Table 4, the rule system $\mathcal{RS}$ contains the rule $\tau(state(l, \overline{Var}), state(\hat{l}, \overline{Var}_{[x \mapsto e]}), param) :- g$. Following the function *oracle* in Table 5, the oracle $\mathcal{O}_\beta$ only contains the rule invocation $\tau(state(l_n, \overline{Var}_n), state(l_{n+1}, \overline{Var}_{n+1}), param)$. This oracle holds since

(a) $\mathcal{RS}$ contains the appropriate rule (see above),

(b) this rule instantiates e with $[\![e]\!]_\theta$ when invoked, and

(c) this rule holds for $l = l_n$, $\hat{l} = l_{n+1}$, $\overline{Var} = \overline{Var}_n$, $\overline{Var}_{[x \mapsto e]} = \overline{Var}_{n+1}$ under valuation θ and $[\![g]\!]_\theta = true$.

5. Let $\beta(\theta)$ be a trace with the prefix δ followed by the transition $\sigma \rightarrow_{!s(v)} \hat{\sigma} \equiv (l, \overline{Var}) \rightarrow_{!s(v)} (\hat{l}, \overline{Var})$. As defined in the step semantics in Table 1 (rule OUTPUT), the appropriate step in the specification is $l \rightarrow_{g \rhd !s(e)} \hat{l}$ with $[\![g]\!]_\theta = true$ and $v = [\![e]\!]_\theta$. According to rule ROUTPUT in Table 4, the rule system $\mathcal{RS}$ contains the rule $s(state(l, \overline{Var}), state(\hat{l}, \overline{Var}), param(e)) :- g$. Following the function *oracle* in Table 5, the oracle $\mathcal{O}_\beta$ only contains the rule invocation $s(state(l_n, \overline{Var}_n), state(l_{n+1}, \overline{Var}_{n+1}), param(v))$ with $v = [\![e]\!]_\theta$. This oracle holds since

   (a) $\mathcal{RS}$ contains the appropriate rule (see above),

   (b) this rule instantiates e with $[\![e]\!]_\theta$ when invoked, and

   (c) this rule holds for $l = l_n$, $\hat{l} = l_{n+1}$, $\overline{Var} = \overline{Var}_n = \overline{Var}_{n+1}$ under valuation θ and $[\![g]\!]_\theta = true$.

6. Let $\beta(\theta)$ be a trace with the prefix δ followed by the transition $\sigma \rightarrow_{?s(v)} \hat{\sigma} \equiv (l, \overline{Var}) \rightarrow_{?s(v)} (\hat{l}, \overline{Var}_{[x \mapsto v]})$. As defined in the step semantics in Table 1 (rule INPUT), the appropriate step in the specification is $l \rightarrow_{?s(x)} \hat{l}$. According to rule RINPUT in Table 4, the rule system $\mathcal{RS}$ contains the rule $s(state(l, \overline{Var}), state(\hat{l}, \overline{Var}_{[x \mapsto Y]}), param(Y))$. Following the function *oracle* in Table 5, the oracle $\mathcal{O}_\beta$ only contains the rule invocation $s(state(l_n, \overline{Var}_n), state(l_{n+1}, \overline{Var}_{n+1}), param(v))$. Thus, this oracle holds since

   (a) $\mathcal{RS}$ contains the appropriate rule (see above),

   (b) this rule instantiates Y with v, when invoked, and

   (c) this rule holds for $l = l_n$, $\hat{l} = l_{n+1}$, $\overline{Var} = \overline{Var}_n$ and $\overline{Var}_{[x \mapsto Y]} = \overline{Var}_{n+1}$ under valuation θ.

Since we assume the constraint-solver to work correctly, this will again be the case if $\beta(\theta) \in [\![M_{Spec}]\!]_{trace}$.

$\underline{\mathcal{RS} \vdash \mathcal{O}_\beta(\theta) \Rightarrow \beta(\theta) \in [\![M_{Spec}]\!]_{trace}}$ We begin with the trace consisting of one transition again.

1. *τ Step:* Let $\mathcal{O}_\beta$ contain only the rule invocation $\tau(state(l_{init}, \overline{Var}_{init}), state(l_1, \overline{Var}_1), param)$. This is the final situation after having applied the function *oracle* to a trace β, which contains one transition $\sigma_{init} \rightarrow_\tau \hat{\sigma} \equiv (l_{init}, \overline{Var}_{init}) \rightarrow_\tau (\hat{l}, \overline{Var}_{init[x \mapsto v]})$ only (see Table 5). The transition itself has been generated from a specification with an edge $l_{init} \rightarrow_{g \rhd x:=e} \hat{l}$, $[\![g]\!]_\theta = true$ and $[\![e]\!]_\theta = v$ (see Table 1, rule ASSIGN). $\mathcal{O}_\beta$ holds under θ because of the rule $\tau(state(l_{init}, \overline{Var}_{init}), state(\hat{l}, \overline{Var}_{init[x \mapsto v]}), param) :- g$ in $\mathcal{RS}$, which holds under θ with $v = [\![e]\!]_\theta$, $\hat{l} = l_1$ and $\overline{Var}_{init[x \mapsto v]} = \overline{Var}_1$. This rule has been generated from M_{Spec} by RASSIGN (see Table 4) for the named edge.

2. *Output action:* Let $\mathcal{O}_\beta$ contain only the rule invocation $s(state(l_{init}, \overline{Var}_{init}), state(l_1, \overline{Var}_1), param(v))$ with $v = [\![e]\!]_\theta$. This is the final situation after having applied the function *oracle* to a trace β, which contains one transition $\sigma_{init} \rightarrow_{!s(v)} \hat{\sigma} \equiv (l_{init}, \overline{Var}_{init}) \rightarrow_{!s(v)} (\hat{l}, \overline{Var}_{init})$

only (see Table 5). The transition itself has been generated from a specification with an edge $l_{init} \rightarrow_{g \rhd !s(e)} \hat{l}$, $[\![g]\!]_\theta = true$ and $[\![e]\!]_\theta = v$ (see Table 1, rule OUTPUT). $\mathcal{O}_\beta$ holds under θ because of the rule $s(state(l_{init}, \overline{Var}_{init}), state(\hat{l}, \overline{Var}_{init}), param(e)) \coloneq g$ in $\mathcal{RS}$, which holds under θ with e being instantiated with $[\![e]\!]_\theta$, $v = [\![e]\!]_\theta$, $\hat{l} = l_1$ and $\overline{Var}_{init} = \overline{Var}_1$. This rule has been generated from M_{Spec} by ROUTPUT (see Table 4) for the named edge.

3. *Input action:* Let $\mathcal{O}_\beta$ contain only the rule invocation $s(state(l_{init}, \overline{Var}_{init}), state(l_1, \overline{Var}_1), param(v))$. This is the final situation after having applied the function *oracle* to a trace β, which contains one transition $\sigma_{init} \rightarrow_{?s(v)} \hat{\sigma} \equiv (l_{init}, \overline{Var}_{init}) \rightarrow_{?s(v)} (\hat{l}, \overline{Var}_{init[x \mapsto v]})$ only (see Table 5). The transition itself has been generated from a specification with an edge $l_{init} \rightarrow_{?s(x)} \hat{l}$ (see Table 1, rule INPUT). $\mathcal{O}_\beta$ holds under θ because of the rule $s(state(l_{init}, \overline{Var}_{init}), state(\hat{l}, \overline{Var}_{init[x \mapsto Y]}), param(Y))$ in $\mathcal{RS}$, which holds under θ with $Y = v$, $\hat{l} = l_1$ and $\overline{Var}_{init[x \mapsto v]} = \overline{Var}_1$. This rule has been generated from M_{Spec} by RINPUT (see Table 4) for the named edge.

Since we only regard a one-transition trace β here, $\beta \in [\![M_{Spec}]\!]_{trace}$ holds in all three cases.

Now we assume an oracle $\mathcal{O}_\delta$ which holds under θ since there exists a $\beta(\theta) \in [\![M_{Spec}]\!]_{trace}$. In the general step, we add a rule invocation to the oracle to complete $\mathcal{O}_\beta$ for trace β with its prefix δ. We then show that $\beta(\theta) \in [\![M_{Spec}]\!]_{trace}$.

4. *τ Step:* Let $\mathcal{O}_\beta$ contain the rule invocations for $\mathcal{O}_\delta$ and one additional rule invocation $\tau(state(l_n, \overline{Var}_n), state(l_{n+1}, \overline{Var}_{n+1}), param)$. This is the final situation after having applied the function *oracle* to a trace β, which contains the transition $\sigma \rightarrow_\tau \hat{\sigma} \equiv (l, \overline{Var}) \rightarrow_\tau (\hat{l}, \overline{Var}_{[x \mapsto v]})$ only (see Table 5). The transition itself has been generated from a specification with an edge $l \rightarrow_{g \rhd x:=e} \hat{l}$, $[\![g]\!]_\theta = true$ and $[\![e]\!]_\theta = v$ (see Table 1, rule ASSIGN). $\mathcal{O}_\beta$ holds under θ because of the rule $\tau(state(l, \overline{Var}), state(\hat{l}, \overline{Var}_{[x \mapsto v]}), param) \coloneq g$ in $\mathcal{RS}$, which holds under θ with $v = [\![e]\!]_\theta$, $l = l_n$, $\hat{l} = l_{n+1}$, $\overline{Var} = \overline{Var}_n$ and $\overline{Var}_{[x \mapsto v]} = \overline{Var}_{n+1}$. This rule has been generated from M_{Spec} by RASSIGN (see Table 4) for the named edge.

5. *Output action:* Let $\mathcal{O}_\beta$ contain only the rule invocation $s(state(l_n, \overline{Var}_n), state(l_{n+1}, \overline{Var}_{n+1}), param(v))$ with $v = [\![e]\!]_\theta$. This is the final situation after having applied the function *oracle* to a trace β, which contains one transition $\sigma \rightarrow_{!s(v)} \hat{\sigma} \equiv (l, \overline{Var}) \rightarrow_{!s(v)} (\hat{l}, \overline{Var})$ only (see Table 5). The transition itself has been generated from a specification with an edge $l_{init} \rightarrow_{g \rhd !s(e)} \hat{l}$, $[\![g]\!]_\theta = true$ and $[\![e]\!]_\theta = v$ (see Table 1, rule OUTPUT). $\mathcal{O}_\beta$ holds under θ because of the rule $s(state(l, \overline{Var}), state(\hat{l}, \overline{Var}), param(e)) \coloneq g$ in $\mathcal{RS}$, which holds under θ with e being instantiated with $[\![e]\!]_\theta$, $v = [\![e]\!]_\theta$, $l = l_n$ $\hat{l} = l_{n+1}$ and $\overline{Var} = \overline{Var}_n = \overline{Var}_{n+1}$. This rule has been generated from M_{Spec} by ROUTPUT (see Table 4) for the named edge.

6. *Input action:* Let $\mathcal{O}_\beta$ contain only the rule invocation $s(state(l_n, \overline{Var}_n), state(l_{n+1}, \overline{Var}_{n+1}), param(v))$. This is the final situation after having applied the function *oracle* to a trace β, which contains one transition $\sigma \rightarrow_{?s(v)} \hat{\sigma} \equiv (l, \overline{Var}) \rightarrow_{?s(v)} (\hat{l}, \overline{Var}_{[x \mapsto v]})$ only (see Table 5). The transition itself has been generated from a specification with an edge $l \rightarrow_{?s(x)} \hat{l}$ (see Table 1, rule INPUT). $\mathcal{O}_\beta$ holds under θ because of the rule $s(state(l, \overline{Var}), state(\hat{l}, \overline{Var}_{[x \mapsto Y]}), param(Y))$ in $\mathcal{RS}$, which holds under θ with $Y = v$, $l = l_n$, $\hat{l} = l_{n+1}$, $\overline{Var} = \overline{Var}_n$ and $\overline{Var}_{[x \mapsto v]} = \overline{Var}_{n+1}$. This rule has been generated from M_{Spec} by RINPUT (see Table 4) for the named edge.

Since the prefix δ of β holds under θ and there exists one of the transitions named above, which also holds under θ, $\beta \in [\![M_{Spec}]\!]_{trace}$ holds. □

The algorithm *NewPassTrace* plays a crucial role in trace selection for test execution. It is not only refered to by the algorithm *SelectAndExecTest*, but also by *ExecTest*, which actually executes a particular test trace. *NewPassTrace* is depicted in Figure 4.

Algorithm 2 (NewPassTrace$(\delta, \theta, M_{TC}^{\top}) : (\beta, \theta') \in [\![M_{TC}^{\top}]\!]_{\mathsf{Pass}} \times \{Var_{symb} \to D\}$*).*

```
1  β := selectFirst(δ, [[M_TC^⊤]]_Pass);
2  while β ≠ no_trace
3    O_β := oracle(0, β)
4    θ' := solve(RS, O_β, ⌊θ⌋_δ)
5    G := O_β(θ')
6    if < G, true > is satisfiable
7      then return (β, θ');
8      else β := selectNext(δ, [[M_TC^⊤]]_Pass);
9    fi
10 elihw
11 return no_solution;
```

Figure 4: Pass trace selection procedure

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 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

An original system may behave nondeterministically, for instance, if it not only evaluates data being sent by the test case but also from other sources, or if the specification is more general than its (refined) implementation. 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. When that happens, the execution of the test case has to be adapted to the new situation dynamically.

Let β be a Pass-trace of $M_{TC}^{\top}$, 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 works as described in the recursive algorithm in Figure 5. 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*, that means no further step has been found in the test trace, 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 nearly equally, 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 the executed trace prefix 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 in 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.

Algorithm 3 *(ExecTest*$(\beta, \theta, \delta, M^{\top}_{TC}) : verdict \in \{$None, Pass, Inconc, Fail$\}$*).*

```
1   step := next(β, δ);
2   case step
3     no_step :
4       if |δ| > 0
5         then setVerdict(Pass);
6         else setVerdict(None);
7       fi
8     τ : ExecTest(β, θ, add(δ, step), M^⊤_TC);
9     !s(X): sendToIUT(s([[X]]_θ));
10      ExecTest(β, θ, add(δ, step), M^⊤_TC);
11    ?s(X): receiveFromIUT(sig(Y));
12      if sig = s ∧ [[Y]] = [[X]]_θ;
13        then ExecTest(β, θ, add(δ, step), M^⊤_TC);
14        else
15          δ' := add(δ, sig(Y));
16          O_δ' := oracle(0, δ');
17          G_δ' := O_δ'(⌊θ⌋_δ[X↦[[Y]]]);
18          if ¬satisfiable(< G_δ', true >)
19            then setVerdict(Fail);
20            else
21              (β', θ') := NewPassTrace(δ', ⌊θ⌋_δ[X↦[[Y]]], M^⊤_TC);
22              if (β', θ') = no_solution
23                then setVerdict(Inconc);
24                else ExecTest(β', θ', δ', M^⊤_TC);
25              fi
26          fi
27      fi
28  esac
```

Figure 5: Test execution procedure

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 gnerated 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 $\mathcal{RS}_{Spec}$. The trace β is then executed. An already executed prefix of β is trace δ.

In the following, we will first prove some invariants over the algorithm. Then we prove that the test verdict assigned to a test case after execution of the algorithm is sound. All line numbers in the proofs refer to the line numbers in *ExecTest* (Figure 5).

Lemma 5 *(Termination of Test Execution).* Given a finite trace β, the test execution algorithm always terminates, given that the *IUT* is deadlock-free.

Proof. As its first statement, the algorithm *ExecTest* always executes a function *next* on trace β to derive the next step in the test to execute. Given a finite trace β, at the end of this trace the function

returns *no_step*. In this case, the trace has been executed completely and the algorithm terminates with either verdict Pass or None (lines 5 and 6).

The function *next* determines the next step in β by comparing it to its already executed prefix δ. Each step executed during the test is appended to δ before the test execution algorithm is reinvoked. This happens in lines 8, 10, 13 and in line 15 before the reinvocation of *ExecTest* (line 24). In so doing, it is guaranteed that δ always reflects the actual state of test execution and that *next* always returns a correct next step or *no_step* after β has been executed completely. □

Lemma 6 *(Assignment of Verdict).* When the test execution algorithm terminates, it always assigns a verdict.

Proof. The test execution algorithm either completely executes β and then terminates assigning Pass or None (see Lemma 5), or it already terminates in lines 23 assigning Inconc or 19 assigning Fail, resp. In all other cases (lines 8, 10, 13 and 24) it is reinvoked and does not terminate with the actual step. □

Lemma 7. For all $\delta(\theta)$, for which the algorithm does not terminate with a Fail verdict, holds: $\delta(\theta) \in [\![M_{Spec}]\!]_{trace}$.

Proof. Proof by induction.

First step: The initial trace β has been chosen by $NewPassTrace(no\_trace, \emptyset, M_{TC}^{\top})$ for execution. In the invocation of this algorithm, no trace prefix is preselected (parameter *no_trace*) and the possible resulting valuation for the chosen trace has also not been limited (parameter $\emptyset$). The selection made by the Pass trace selection algorithm is the trace β which is per definition (Definition 9) a trace in $[\![M_{Spec}^{\top}]\!]_{trace}$. The trace can be instantiated by θ to $\beta(\theta) \in [\![M_{Spec}]\!]_{trace}$, since *NewPassTrace* ensures $\mathcal{O}_\beta := oracle(0, \beta) \wedge \theta := solve(\mathcal{RS}, \mathcal{O}_\beta, M_{TC}^{\top}) \wedge G := \mathcal{O}_\beta(\theta) \wedge < G, true >$ is satisfiable, which holds iff $\beta(\theta) \in [\![M_{Spec}]\!]_{trace}$ (Lemma 4 and proof). Since $\delta(\theta)$ is a – possibly empty – prefix of $\beta(\theta)$, the aforementioned claim also holds for $\delta(\theta)$.

Inductive step: The test execution algorithms recursively executes $\beta(\theta)$ transition by transition with $\delta(\theta)$ being the prefix, which has already been executed. Taking such an arbitrary transition, we now have to regard the recursive invocation of *ExecTest*, whether $\beta'(\theta') \in [\![M_{Spec}]\!]_{trace}$ still holds if $\beta(\theta) \in [\![M_{Spec}]\!]_{trace}$.

1. First we have to regard the recursive invocation in lines 8, 10 and 13. In all three cases $\beta' = \beta \wedge \theta' = \theta$ so that our claim holds (trivial case). The already executed prefix of β' is $\delta'(\theta')$ in this case, for which of course $\delta'(\theta') \in [\![M_{Spec}]\!]_{trace}$ also holds, since neither β nor θ have changed.
2. In line 24, both $\beta' \neq \beta \wedge \theta' \neq \theta$. In this case the new trace β' to Pass is searched by *NewPassTrace* and executed only, if $\mathcal{O}_{\beta'} := oracle(0, \beta') \wedge \theta' := solve(\mathcal{RS}, \mathcal{O}_{\beta'}, M_{TC}^{\top}) \wedge G := \mathcal{O}_{\beta'}(\theta') \wedge < G, true >$ is satisfiable, which holds iff $\beta'(\theta') \in [\![M_{Spec}]\!]_{trace}$. Thus for the already executed prefix of $\beta'(\theta')$ also holds: $\delta'(\theta') \in [\![M_{Spec}]\!]_{trace}$.

In all cases, where $\beta \notin [\![M_{Spec}]\!]_{trace}$, this is discovered and test execution terminates with a Fail verdict (line 19 and appropriate proof).

□

Lemma 8 *(Soundness of verdict* Fail*).* In case, that the verdict Fail is assigned, for the trace $\delta' = add(\delta, sig(Y))$ holds: $\delta'(\theta_{[X \mapsto [\![Y]\!]]}) \notin [\![M_{Spec}]\!]_{trace}$.

Proof. First of all, the Fail verdict is assigned only in line 19, where input from the IUT is evaluated. It is checked whether the executed trace $\delta' = add(\delta, sig(Y)) \in [\![M_{Spec}]\!]_{trace}$ under $\lfloor\theta\rfloor_{\delta[X\mapsto[\![Y]\!]]}$.

The valuation θ has been precalculated for the whole test sequence β. $\lfloor\theta\rfloor_\delta$ denotes that part of θ, which is relevant for the subtrace δ. Accordingly, $\lfloor\theta\rfloor_{\delta[X\mapsto[\![Y]\!]]}$ denotes the same part of the valuation with X being set to the value of Y.

The trace is valid only, if $G_{\delta'} = \mathcal{O}_{\delta'}(\lfloor\theta\rfloor_{\delta[X\mapsto Y]}) \wedge < G_{\delta'}, true >$ is satisfiable (see Definition 4 and proof). The verdict Fail is set only in those cases, where $< G_{\delta'}, true >$ is not satisfiable and thus $\delta'(\lfloor\theta\rfloor_{\delta[X\mapsto[\![Y]\!]]}) \notin [\![M_{Spec}]\!]_{trace}$. For this reason, the assignment of the Fail verdict is sound. □

Lemma 9 *(Soundness of verdict* Inconc*).* In case, that the verdict Inconc is assigned, for the executed trace δ holds: $\delta \in [\![M_{Spec}]\!]_{trace} \wedge \delta \notin [\![M_{TC}]\!]_{\mathsf{Pass}}$.

Proof. The verdict Inconc is assigned in line 23.

In this case $\delta'(\theta_{[X\mapsto[\![Y]\!]]})$, consisting of the previously executed trace δ and the action under consideration $sig(Y)$ (both under valuation $\theta_{[X\mapsto[\![Y]\!]]}$) is a trace from $[\![M_{Spec}]\!]_{trace}$ ($G := \mathcal{O}_{\delta'}(\lfloor\theta\rfloor_{\delta[X\mapsto[\![Y]\!]]}) \wedge < G, true >$ is satisfiable; cf. Lemma 4) and no further trace to a Pass verdict could be found which results in *NewPassTrace* returning *no_solution*. This means, that either no trace has been found in the test case, or a trace β' has been found, but $\beta' = NewPassTrace(\delta', \lfloor\theta\rfloor_{\delta[X\mapsto[\![Y]\!]]}, M^{\top}_{TC}) \wedge \mathcal{O}_{\beta'} = oracle(0, \beta') \wedge \theta' = solve(\mathcal{RS}, \mathcal{O}_{\beta'}, \lfloor\theta\rfloor_{\delta[X\mapsto[\![Y]\!]]} \wedge G := \mathcal{O}_{\beta'}(\theta') \wedge < G, true >$ is not satisfiable). In both cases, verdict Inconc has to be assigned per definition (see Definition 9). Thus, the assignments of verdict Inconc in the algorithm *ExecTest* are sound. □

Lemma 10 *(Soundness of verdicts* Pass *and* None*).* In case, that the verdict Pass is assigned, the executed trace $\delta(\theta) \in [\![M_{Spec}]\!]_{trace} \wedge \delta \in [\![M^{\top}_{TC}]\!]_{\mathsf{Pass}} \wedge |\beta| > 0$. In case that $|\beta| = 0$, None is assigned.

Proof. The Pass verdict is assigned in line 5 in those cases only, where a trace $\beta(\theta) \in [\![M_{Spec}]\!]_{trace}$ (cf. Lemma 7), with β having been found in $M^{\top}_{TC}$ by *NewPassTrace*, could be executed to its very end without any Fail or Inconc verdicts assigned. Under these conditions and if the executed trace has had at least one transition, then assigning the Pass verdict is sound (see line 5). In those cases, where no transition had been executed, setting the None verdict is sound (see line 6). □

Lemma 11. The assignment of the test verdict to a test trace is sound.

Proof. This lemma immediately results from the three previous proofs. □

7. CEPS Case Study

In this section, we describe the application of our approach to the case study *CEPS*. The **C**ommon **E**lectronic **P**urse **S**pecifications (CEPS) define a protocol for electronic payment using a chip card as a wallet. The specifications consist of the *functional requirements* [5] and the *technical specification* [6]. 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. On the card, the card reader and at the site of the card issuer, software applications are running and communicating with each other. The roles as represented by software components are depicted in Figure 6.

In our work on the case study, we aim to evaluate our test generation process by automatically generating parameterizable test cases from a μCRL specification for the card application `CEPCardApp` in Figure 6. Our approach starts from a formalized version of the technical specification of the CEPS system[3], which we realized as a μCRL specification. In this specification, all input variables are substituted by the abstract value $\top$. By doing so, the problem of state space explosion in the next steps of the process is avoided w.r.t. the system's interaction with environment. Afterwards, an LTS is generated from this abstracted specification. Further, this LTS is used for the actual test case

[3] see: `http://www.irisa.fr/vertecs/Equipe/Rusu/FME02/ceps.if`

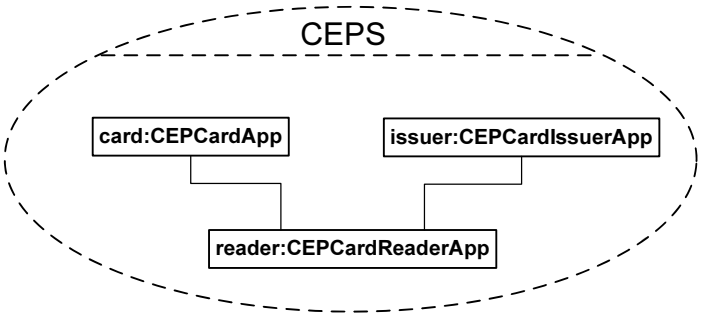

Figure 6: Collaborating System Roles in CEPS

generation. The generation process itself is guided by the *test purpose*. It describes a scenario, which is the focus of the later test cases. This scenario is derived from the system's functional requirements documents. From the abstract specification LTS and the test purpose a set of abstract test cases are generated. Using the original specification, these test cases are parameterized based on a rule system for actual data selection.

Here, we consider the card application `CEPCardApp` as the system under test (SUT). Doing so, the card reader application (`CEPCardReaderApp`) must be the stimulating testing system. The card issuer (`CEPCardIssuerApp`) can be simulated by the card reader application since there is no direct communication between the issuer and the card (see Figure 6). Testing the card application thereby means stimulating it with messages and verifying the received responses whether they are plausible. For the derivation of the test purpose, we regard the use case *load transaction*. The use case *load transaction* is described in [26]. In the following description, which is partially based on an existing NTIF specification (courtesy of the VASY team at INRIA Rhône-Alpes, cf. [16]) of the purse card specification, all messages between the card reader application and the card are named after the NTIF card specification, while the messages between card reader application and the card issuer are named after [26], since there is no counterpart in the NTIF specification.

In a first step, the reader initializes the card by sending a *CepCommand* message, parameterized with *LOADINIT*, and receiving a response *CepReply*, telling that the initialization was successful (code *x9000*). Then the real transaction starts. The card reader application requests money from the card issuer (*Load* message) and gets it with a response *RespL*. This money is then credited on the card (another *CepCommand* message, this time parameterized with *LOADCREDIT*). The response *CepReply* from the card is accepted by the card reader application and the card issuer is informed via a *Comp* message. The interactions in this use case are depicted in Figure 7. Sets of uninteresting parameters in the messages are marked with one hyphen.

In the following subsections, we discuss the application of our test generation process, namely the generation of abstract test cases and the concretization and parameterization of these test cases, to the case study. Afterwards, we outline related research of other groups on CEPS.

Test Case Generation

In our case, test case generation is based on the model of the SUT given as a μCRL specification. This specification is abstracted and an LTS is generated which is then combined with the test purpose, given as an LTS, too. Depending on the states, reached in the test purpose (*refuse* or *accept* states), verdicts are assigned to the according traces in the resulting abstract test case (cf. [3]). Test data, and thus the possibility to identify spurious traces, is introduced by constraint-solving.

For test case generation, we first realize the NTIF specification in μCRL and simplify it. We do not want to test the logging activities of the SUT (as can already be seen in our test scenario in Figure 7), so that this part is not modelled. We took a part of the CEPS and removed several interface variables

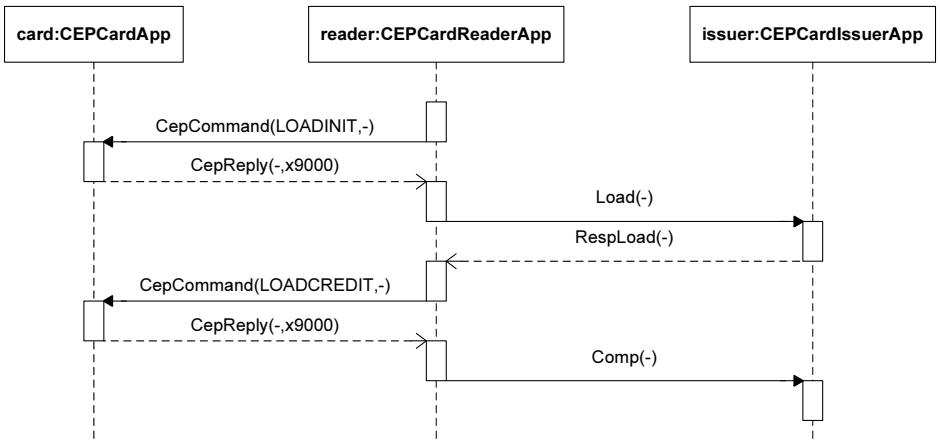

Figure 7: Interactions in the test purpose scenario

to reduce the size of data structures. By doing so, `CommandType`, the data structure sent with the action `CepCommand`, is reduced from 15 elements to 5, `ReplyType`, the data structure for replies form the SUT, from 22 to 16 elements.

The internal variables of the SUT, which also includes internal arrays, are left untouched. These internal arrays are necessary for CEPS to, for instance, manage slots on the purse card for different currencies. For this reason, we do not have to only handle arrays of data *elements*, but also arrays of data *structures*. A purse card has three slots for different "reference currencies" implemented as an array with three elements. Each element is a data structure of two fields, describing one such reference currency. 16 further slots on the card refer to the reference currencies, storing amounts of money in each of these currencies. Each element of this array is again a data structure of five fields, leading to brutto 80 variables for this array. A third array then stores a boolean value for each of the elements of the second array, telling whether this element has been "reported" or not. Due to the arrays and data structures involved, the 44 global variables for the purse card represent brutto 207 single data values – local variables not yet included. During constraint solving, which is described later in this section, the number of these internal variables that has to be handled, alone increases with each step of the abstract test case, ending at approx. 7245 single data values for the smaller of the two examples (and 118818 single values for the larger one).

In a next step, the specification is abstracted. Thereby, for every datatype, an additional abstracted datatype containing $\mathbb{T}$ (`TT_<datatype>`) and the lifting function κ (`known:<datatype>-><abstracted datatype>`) are introduced. An example is given, showing the datatype `CommandCodeType` and its abstracted variant `CommandCodeType_abstr`:

```
sort CommandCodeType

func
  NONE:->CommandCodeType
  SPECIFIC:->CommandCodeType
  ALLSLOTS00:->CommandCodeType
  ALLSLOTS01:->CommandCodeType
  REFCURR:->CommandCodeType
  LOGINQ00:->CommandCodeType
  LOGINQ01:->CommandCodeType
  LOADINIT:->CommandCodeType
  LOADCREDIT:->CommandCodeType
  ERROR_INQ:->CommandCodeType

map
```

```
  eq:CommandCodeType#CommandCodeType->Bool

rew
  eq(NONE,NONE) = T
  eq(NONE,SPECIFIC) = F
...

sort CommandCodeType_abstr

func
  TT_CommandCodeType :                  -> CommandCodeType_abstr
  known              : CommandCodeType -> CommandCodeType_abstr

map eq  : CommandCodeType_abstr#CommandCodeType_abstr -> Bool_abstr

var x,y : CommandCodeType
rew
  eq(TT_CommandCodeType,TT_CommandCodeType) = TT_Bool
  eq(TT_CommandCodeType,known(x))           = TT_Bool
  eq(known(x),TT_CommandCodeType)           = TT_Bool
  eq(known(x),known(y))                     = known(eq(x,y))
```

All process global parameters are defined as parameters of the abstracted datatypes. They are initialized with the original values from the original specification, lifted to the abstracted data types. For every summand in the specification, those input variables, which are of type *Bool* or *Nat* are abstracted to $\mathbb{T}$, all others are at the moment left untouched. As an example, we show one such summand before and after abstraction:

```
% before abstraction
...
sum(mInquiry2:CommandType,CepCommand(mInquiry2).
    X(x2p1(x2p0(one)),...,pNT_Limit,...,vNT,vDeactivated,vLocked,
    getLoadAmt(mInquiry2),0,getCurrency(mInquiry2),0,...,getNewBalMax(mInquiry2),
    ...,mInquiry2,...)
 <|and(eq(s0,x2p0(x2p0(x2p1(x2p0(one))))),and(and(and(eq(getCommand(mInquiry2),
     LOADINIT),
     not(not(lt(vNT,pNT_Limit)))),not(vDeactivated)),not(vLocked)))|>delta)+
...

% after abstraction
...
CepCommand(TT_CommandType).
% alternative: CepCommand(commandData(TT_CommandCodeType,TT_Nat,TT_TxTypeType,TT_Nat,
    TT_Nat)).
X(x2p1(x2p0(known(one))),...,pNT_Limit,...,vNT,vDeactivated,vLocked,
  getLoadAmt(TT_CommandType),known(0),getCurrency(TT_CommandType),known(0),...,
  getNewBalMax(TT_CommandType),...,mInquiry2,...)
  <|may(and(eq(s0,x2p0(x2p0(x2p1(x2p0(known(one))))))),and(and(and(eq(getCommand(
      TT_CommandType),LOADINIT),
     not(not(lt(vNT,pNT_Limit)))),not(vDeactivated)),not(vLocked))))|>delta+
...
```

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

For the first mutent, 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 (operating system: SuSE Linux 9.3, kernel 2.6.11.4-20a-default). 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 (operating system: Redhat Linux Fedora Core 1, kernel 2.4.22-1.2199.nptl).

For the second mutant, whose abstracted specification led to an LTS of 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 (the limitation to 100 steps led to identical results as without any limits) on the single CPU PC.

Test Case Parameterization

For test case parameterization, first a trace to a Pass verdict is selected from the generated abstract test case. We select only this kind of traces for the moment, since we are not interested in distinguishing between Inconc and Fail verdicts when executing a test. For this reason, we only want to deliver a valuation for the variables appearing in the test case, that leads to a Pass verdict, not to Inconc or Fail. Some parts of such a trace are depicted in Figure 8 and printed in Appendix B.2.

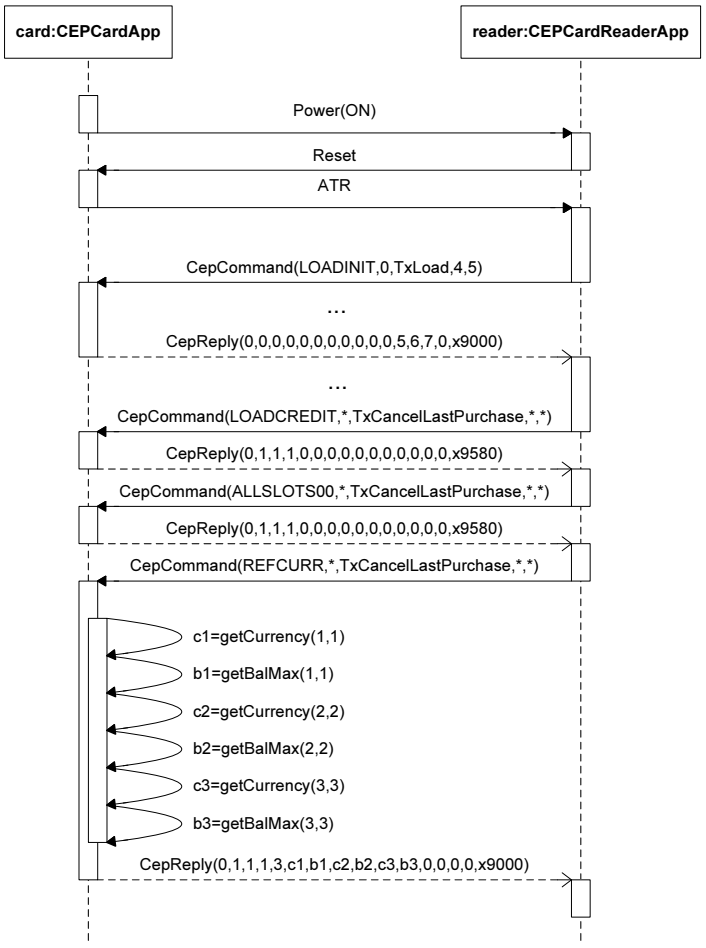

Figure 8: Test Case Example for CEPS

This test case is derived from the specification of the SUT together with the test purpose for the

scenario in Figure 7. It is easy to recognize, that the test case contains more actions than the test purpose. The reason is, that the test purpose only sketches the main focus while the test case must also cover initialization actions like `Power(ON)` as well as preparing actions for the test purpose action (for instance, iteration through an array to a certain position). The actions from the test case are later relevant for the determination of the test oracle.

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 SUT 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 SUT.

The basic rule system consists of two parts. In the first part, a meta language is defined based on the functions of the original μCRL specification. Below, the definition of the datatype *Bool* as shown in Figure 1 is printed:

```
cT.                                    % Constructor "true"
cF:- fail.                             % Constructor "false"
tBool(X):- X.                          % Expressions of sort Bool as
                                       % Prolog expressions
and(X,Y):- X,Y.                        % Meta function and() for Prolog "and"
and(tBool(V0),tBool(V0)):- tBool(V0). % Realization of mCRL and()
and(tBool(cF),tBool(V0)):- tBool(cF). % functions as Prolog rules.
and(tBool(V0),tBool(cF)):- tBool(cF).
and(tBool(V0),tBool(cT)):- tBool(V0).
and(tBool(cT),tBool(V0)):- tBool(V0).
...
```

The realization of this datatype is special, since here the datatype *Bool* from μCRL is not only mapped to Prolog, but is also given semantics. This explains the existence of definitions of rules for the constructor `T` and `F` (here: `cT` and `cF` to conform to lower-case writing for Prolog rules). `T` is thereby realized as a Prolog rule, which holds, while `F` is realized as a failing rule. (The absence of the latter rule would have the same effect.) Prolog is not as strongly typed as, for instance, μCRL. This complicates polymorphism, that means finding the right function not only by its name but also by the types of its parameters. For this reason, all parameters for the Prolog rules are enriched with their type information like `tBool`. This can be seen in the five realizations of the μCRL `and`. Since again these rules get Prolog semantics, a mapping from `tBool(<expression>)` to a Prolog Boolean expression, which can be further evaluated, is given (3rd line). The realization of *Bool* is especially complex, since it is the basis of all Prolog work on the specification rules. All other rules are conceptually less complex, since they simply use the existing *Bool* basic.

The second part of the rule system defines rules for each of a linearized μCRL specification's summands. As an example, one rule is shown in appendix B.3.

A rule has three parameters. The first two define changes of global variables during execution of the action, which belongs to this rule in the specification. In this case, it is an assignment to the variable `mInquiry`. This results in a replacement of the Prolog variable `VmInquiry` (all variable names begin with a capital `V` in our Prolog realization) from the first *global*-parameter by `VmInquiry2` in the second *global*-parameter. Since this is the effect of an input from environment, `VmInquiry2` also appears in the *local*-parameter of the rule. The part after `:-` defines the condition, under which the action can be executed. In the first operand of the following (Prolog) conjunction, `getCommand`, the command code type (in this case *LOADINIT*) is extracted from the variable `VmInquiry2`. The second operand defines an exact reproduction of the original condition from the specification using the meta language. In detail, the control location, in which the transition starts, is checked to be 20 ($= ((1 \cdot 2 \cdot 2) + 1) \cdot 2 \cdot 2$), the variable `VvDeactivated` is *true* and the extracted command code type is *LOADINIT*.

Having built up the rule system, we can ask queries to it. In the existing implementation, all queries

which form one trace in the test case are combined to a test oracle. One such oracle (oracle 0, the only one for our trace) is partially printed in appendix B.4 as an example.

The parameters of the oracle are all those variables, which are input from the SUT's environment or relevant output from the SUT. The body of the oracle rule is given by the conjunction of the rules, which are relevant for the test case under consideration. In the first rule, here `power`, the global variables of the SUT are initialized with the values defined in the specification. All changes in the configuration of the global variables' valuations are from then on collected in one variable and passed to the next rule in the conjunction. Local variables are always collected in the third parameter of each rule; if there exist no local variables for a rule, the structure `local` stays empty (see `reset` or `aTR`). The last change of the global valuation (rule `cepReply(G89, _, local(...))`) is set to the Prolog *don't-care* parameter, since we are not interested in further steps of the SUT and thus in global variables. Executing this oracle leads to a result, describing the possible valuations for input and output variables of the system. Parts of it are shown in appendix B.5.

For a variable, either fixed values can be set (like for `Power`) or the concrete value of the variable can be chosen from a range and is not fixed to one value. In this case, the oracle shows a variable instead of a fixed value (e.g. third, fourth and fifth element of the data structure for `CC2` in the example in appendix B.5). We have to annotate, that the proposal of the constraint solver for variable values is not bound to those values, which are used in the test purpose (not all values there are don't-care, cf. the command code types in `CepCommand`). The constraint solver only produces a solution for the given sequence of actions. If the solution set shall be reduced to the values from the test purpose, these must be fixed when invoking the test oracle.

Within the test oracle definition, the initial values for the process are set as parameters of the first rule in the conjunction. These values can be left variable, if we want to adjust them on-the-fly at the beginning of test execution. In this case, we would have defined them as $\mathbb{T}$ values before generating the parameterizable test case. This would have led to more transitions in the model of the abstracted SUT. So, we decided to do the experiment leaving the initialization of the process at its original values.

The first parameter of the first rule defines the initial control location of the process as of the specification. All other parameters define initial values for the global variables. The card application has four groups of global variables, which are the *parameters of the card application* like issuer's and card identification numbers or the card's expiration date, the *static storage* of the card, its *working storage* and *messages*, which are exchanged between the card and the card reader. Since the last two groups have to be initialized, simply because they exist, but their initial values are not relevant at all, we only concentrate on some aspects of the *static storage*, the card's slots.

For our case, the card has 15 slots of which the first 3 are used. Each of them is initialized to be in use (*true*) and that the currency for its slot cannot be changed (also *true*). Furthermore, each slot gets a currency identifier and a balance in this currency. In our initialization, this actual balance is equal to the maximal balance possible for this card slot. These three currencies described here are at the same time the reference currencies of this card. All the other card slots are initialized with placeholder values (0 and false). The array of reported slots, the next parameter, is completely initialized with *false* since at the start time of the process, which we are describing here, no array slots can have been reported to anyone.

A limitation for Prolog rule systems and also constraint solving in general is the limitation of computer memory. Already the CEPS example leads to a vast amount of variables in our test oracle, for the limited test case 22 input and 3960 internal variables which must be introduced and which are in parts themselves data structures of, for instance, 16 elements each (e.g. `ReplyType` or the arrays; also see a previous paragraph of this section for the number of variables in the examples). At this point, the approach of constraint solving for test data selection can relatively easily reach certain limits. However, in our sample test case it is possible to calculate solutions for the constraints, under which the test case can be executed.

Regarding the actual calculation of solutions for the constraint system, again the question comes up, if a calculation in advance to the test execution or on-the-fly is the better choice. In this case,

the on-the-fly solution is more promising. Calculating a complete constraint solution in advance to the test execution fixates it to exactly one trace through the SUT model. But if the SUT reacts in a nondeterministic way, this approach will lead to faulty Fail or Inconc verdicts, if the SUT chooses another trace to Pass than the one precalculated. Calculating all possible traces to Pass would also be at least very inefficient, so that an on-the-fly calculation, adapted to the actual execution trace, is the better solution here. However, test data derived from on-the-fly constraint solving can still be collected and possibly reused in later test reexecutions. The considerations of test execution and, connected to that, data selection are an integral part of our future work.

8. Conclusion

Related Works

Test Generation with Data Abstraction The closest to our approach is *symbolic test generation* [9, 15, 23, 31, 37]. This method works directly on higher-level specifications given as input-output symbolic transition systems (IOSTSs) without enumerating their state space. Given a test purpose and a specification, their product is built. The coreachability analysis is in these cases over-approximated by Abstract Interpretation [10].

The purpose and usage of abstraction techniques in our approach is conceptually different from the one of symbolic test generation, since we use a data abstraction that mitigates infinity of external data. This enables us to use existing enumerative test generation techniques for the derivation of abstract test cases which are then 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 SUT's behavior. They both also employ constraint solving to choose a single testing strategy during test execution, so that more case studies are needed to conclude which approach is more suitable for which class of systems.

The rule systems, we generate to determine test data, are comparable to [30], where also constraint rules are generated encoding the visible inputs and outputs, guards and internal state changes. These rules are then used to generate a set of test cases by transforming a system specification in its whole into Prolog. However, in our case, test cases are already present.

CEPS Case Study CEPS has been the basis for several case studies in which different aspects are analyzed. On the one hand, they are related to security aspects concretely regarding the standard, on the other hand CEPS serves as a sandbox for experiments with test generation in general.

Security aspects of the standard are analyzed on the level of models. In [26] an extension of UML, *UMLsec*, is exemplarily applied to CEPS for that purpose. This approach, named *security modelling*, is driven further in [25] by providing a formal model for a subset of CEPS and applying model checking to it. In [35] the specification-based test of security-critical protocols like CEPS is discussed. The approach is thereby a combination of scenario- and specification-based test generation, which is guided by mutants of the SUT's specification and attack scenarios. Another test sequence generation approach, this time using propositional logic solvers, is worked out in [24].

An approach for the specification-based generation of symbolic tests and test oracles is introduced by [7]. Here, both the specification of the SUT and the test purpose are expressed as labeled transition systems but are processed symbolically with respect to variables, message parameters and value-passing to avoid the well-known problem of state-space explosion. Regarding test case generation, this approach probably leads to the problem described for test case generation with on-the-fly generation of a non-reusable state space.

The tool *STG*, developed for this approach, is presented in [8] and [9] with the CEPS case study. This tool is also applied to CEPS in [32], but here to extract components whose correctness is verified by afterwards using a theorem prover.

Conclusion and Future Work

In this paper, an approach for the automatic generation of conformance test cases was discussed. The approach is strongly based on the idea of applying data abstraction to receive a specification with a manageable state space as the starting point for the generation process. The correctness of this approach has been proven and its applicability has been tested in the scope of a case study. Furthermore, an approach for data selection based on constraint-solving has been introduced. It could also successfully be applied to the case study. Some of the results have been published in [4].

Up to this point, only platform-independent test models and data constraints are generated by our approach, starting from a formal (process-algebraic) specification. This leaves two interesting directions for further work. On the one hand, starting test generation from UML models would allow the integration of our approach into existing software development processes. On the other hand, the generation of actual test code has not yet been realized. Here, a generator for TTCN-3 parameterizable test cases is being developed. A problem of special interest is thereby the integration of data selection into these test cases, for instance by online constraint solving during test execution as proposed in this paper.

A. Test Execution Algorithm

In this section, the algorithms from Section 6 are shown graphically. Due to technical reasons, $\mathbb{T}$ has had to be replaced by $*$ in the figures.

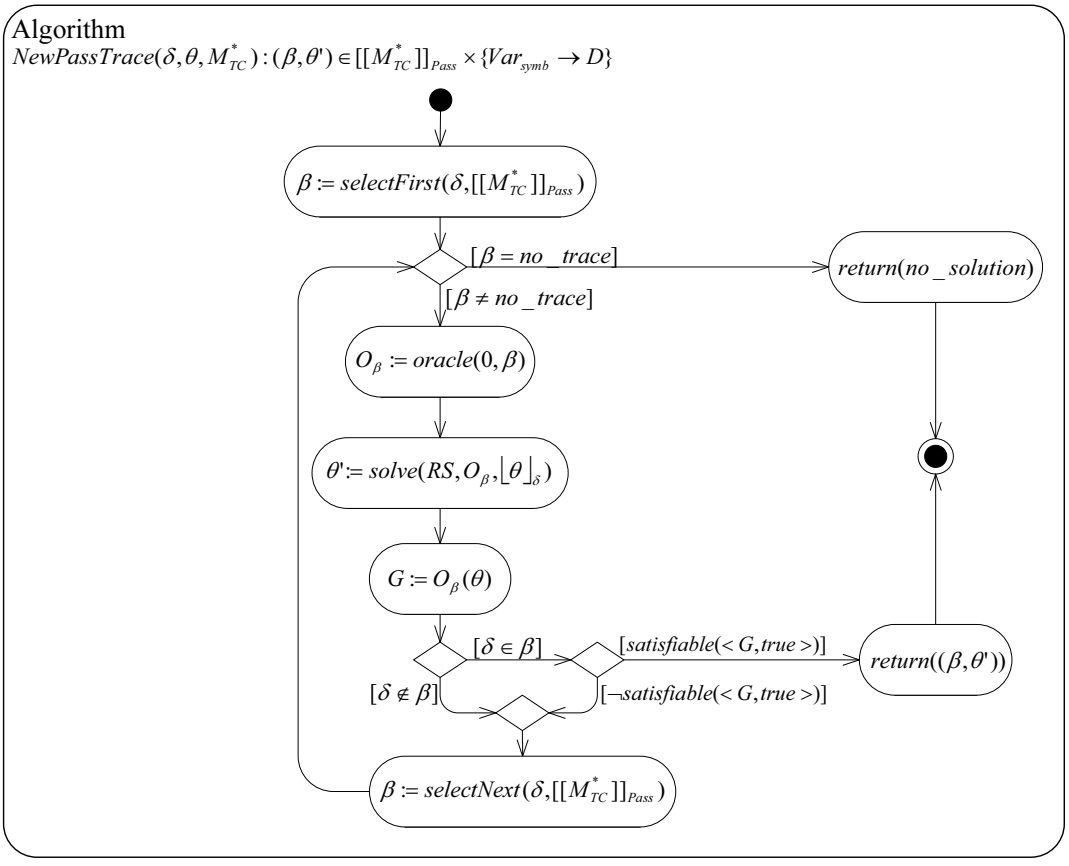

Figure 9: Algorithm *NewPassTrace*

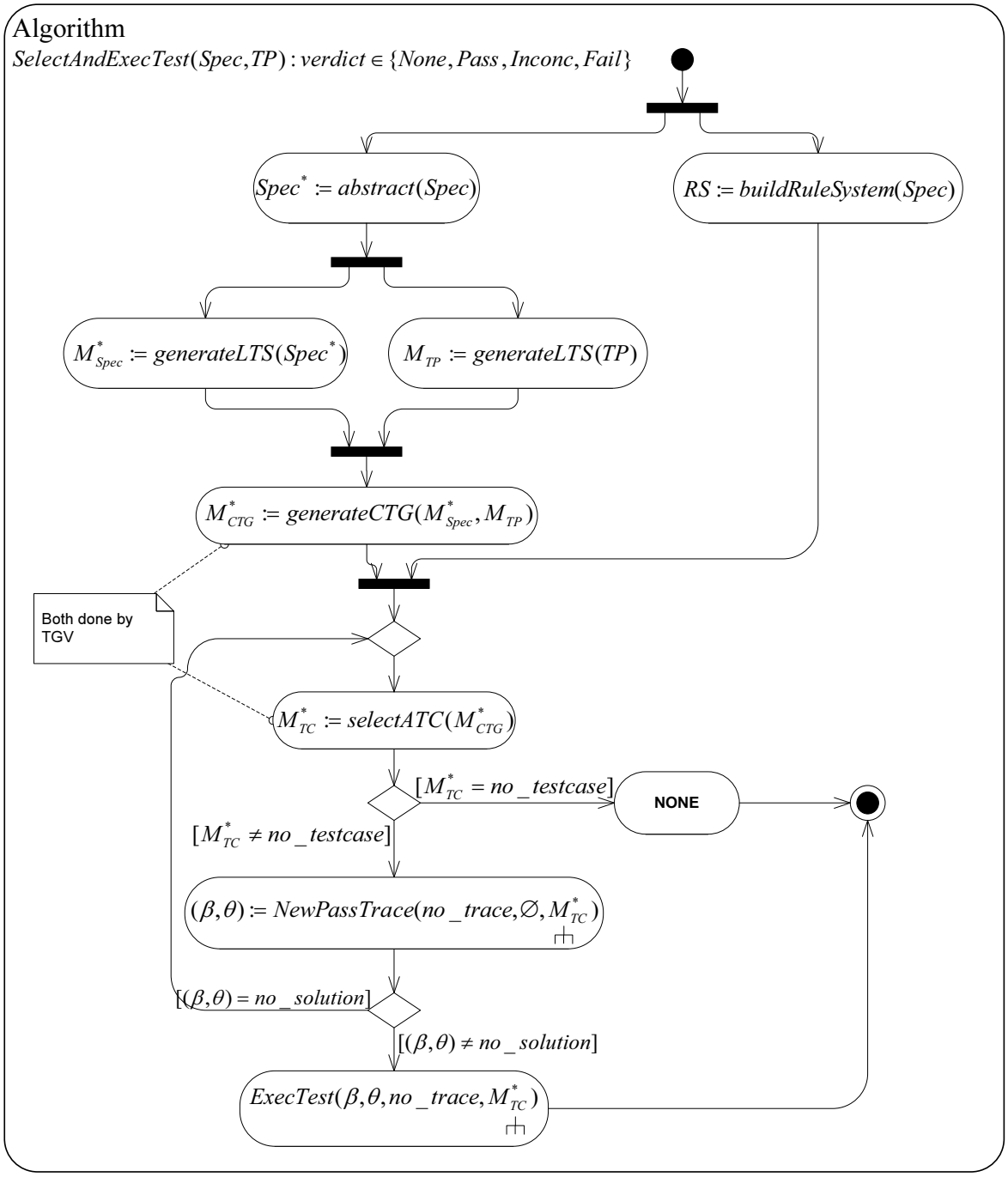

Figure 10: Algorithm *SelectAndExecTest*

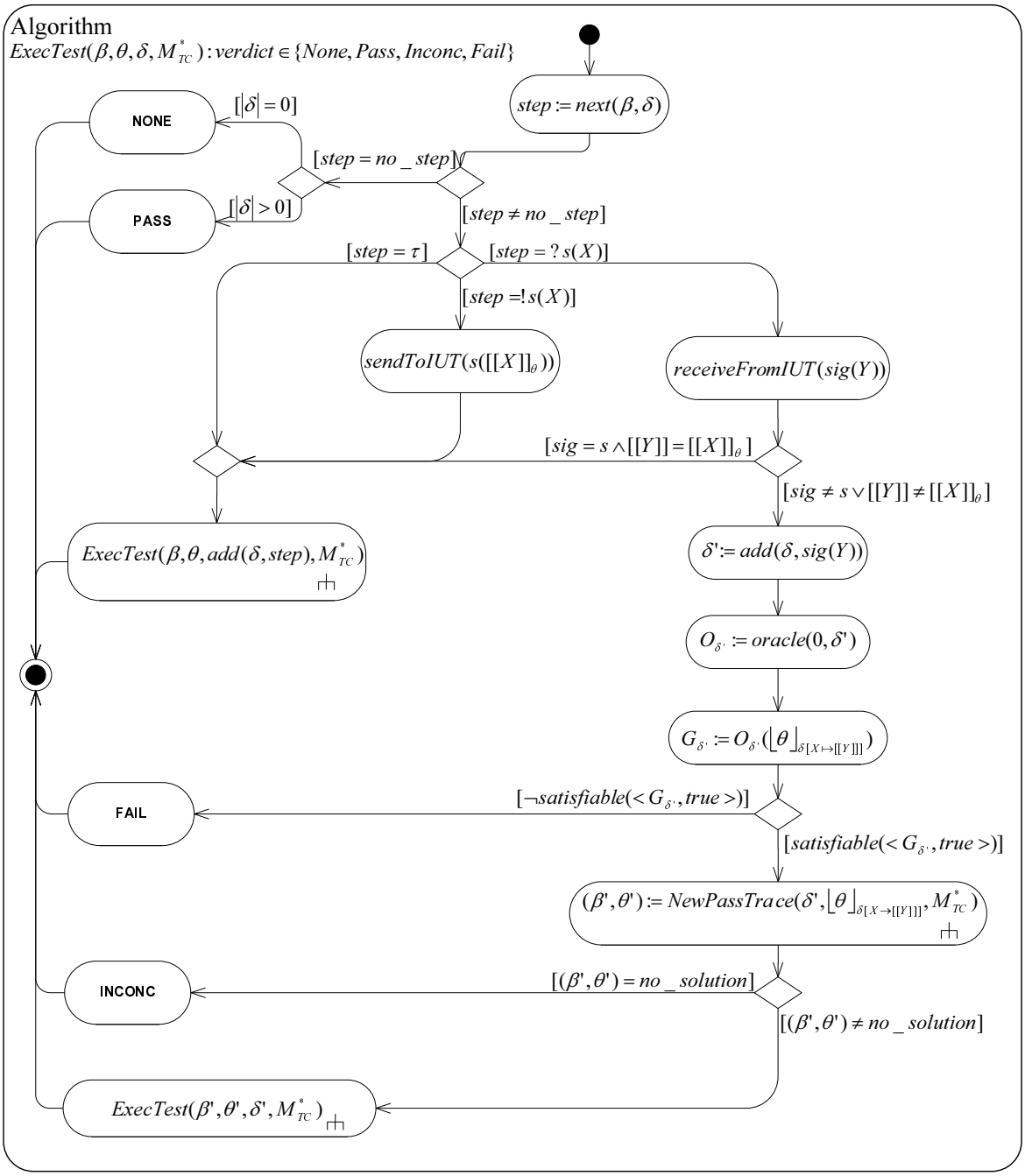

Figure 11: Algorithm *ExecTest*

B. Selected Codes

B.1 Abstraction of CommandCodeType

```
sort CommandCodeType

func
  NONE:->CommandCodeType
  SPECIFIC:->CommandCodeType
  ALLSLOTS00:->CommandCodeType
  ALLSLOTS01:->CommandCodeType
  REFCURR:->CommandCodeType
  LOGINQ00:->CommandCodeType
  LOGINQ01:->CommandCodeType
  LOADINIT:->CommandCodeType
  LOADCREDIT:->CommandCodeType
  ERROR_INQ:->CommandCodeType

map
  eq:CommandCodeType#CommandCodeType->Bool

rew
  eq(NONE,NONE) = T
  eq(NONE,SPECIFIC) = F
...

sort CommandCodeType_abstr

func
  TT_CommandCodeType :                  -> CommandCodeType_abstr
  known              : CommandCodeType -> CommandCodeType_abstr

map eq  : CommandCodeType_abstr#CommandCodeType_abstr -> Bool_abstr

var x,y : CommandCodeType
rew
  eq(TT_CommandCodeType,TT_CommandCodeType) = TT_Bool
  eq(TT_CommandCodeType,known(x))           = TT_Bool
  eq(known(x),TT_CommandCodeType)           = TT_Bool
  eq(known(x),known(y))                     = known(eq(x,y))
```

B.2 Test Case Example

```
---
Power(known(ON))
Reset
ATR
CepCommand(commandData(known(LOADINIT),known(0),known(TxLoad),known(succ(succ(succ(
    succ(0))))),known(succ(succ(succ(succ(succ(0))))))))
...
CepReply(replyData(known(0),known(0),known(0),known(0),known(0),known(0),known(0),
    known(0),known(0),known(0),known(0),known(succ(succ(succ(succ(succ(0)))))),known(
    succ(succ(succ(succ(succ(succ(0)))))),known(succ(succ(succ(succ(succ(succ(0)
    ))))))),known(0),known(x9000)))
...
CepCommand(commandData(known(LOADCREDIT),TT_Nat,known(TxCancelLastPurchase),TT_Nat,
    TT_Nat))
CepReply(replyData(known(0),known(succ(0)),known(succ(0)),known(succ(0)),known(0),
    known(0),known(0),known(0),known(0),known(0),known(0),known(0),known(0),
    known(0),known(x9580)))
CepCommand(commandData(known(ALLSLOTS00),TT_Nat,known(TxCancelLastPurchase),TT_Nat,
    TT_Nat))
CepReply(replyData(known(0),known(succ(0)),known(succ(0)),known(succ(0)),known(0),
    known(0),known(0),known(0),known(0),known(0),known(0),known(0),known(0),
    known(0),known(x9580)))
```

```
CepCommand(commandData(known(REFCURR),TT_Nat,known(TxCancelLastPurchase),TT_Nat,TT_Nat
    ))
CepReply(replyData(known(0),known(succ(0)),known(succ(0)),known(succ(0)),known(succ(
    succ(succ(0)))),getCurrency(refCurr(known(succ(0)),known(succ(0)))),getBalMax(
    refCurr(known(succ(0)),known(succ(0)))),getCurrency(refCurr(known(succ(succ(0))),
    known(succ(succ(0)))),getBalMax(refCurr(known(succ(succ(0))),known(succ(succ(0)))
    )),getCurrency(refCurr(known(succ(succ(succ(0)))),known(succ(succ(succ(0))))),
    getBalMax(refCurr(known(succ(succ(succ(0)))),known(succ(succ(succ(0))))),known(0)
    ,known(0),known(0),known(0),known(x9000)))
---
```

B.3 Summand Rule

```
...
cepCommand(global(tState(Vs0),tNat(VpSlotCount),tNat(VpRefCurCount),
                  tNat(VpLogSize),tNat(VpNT_Limit),tNat(VvIssId),
                  ..., tBool(VvDeactivated) ,... % lots of parameters
                  tCommandType(VmInquiry),tReplyType(VmSlotInfo),
                  ... % and even some more
                  tNat(VmRefCurrCurrency),tNat(VmRefCurBalMax)),
           global(tState(cx2p1(tState(cx2p1(tState(cone))))),
                  tNat(VpSlotCount),tNat(VpRefCurCount),
                  tNat(VpLogSize),tNat(VpNT_Limit),tNat(VvIssId),
                  ..., tBool(VvDeactivated) ,... % unchanged parameters
                  tCommandType(VmInquiry2),tReplyType(VmSlotInfo),
                  ... % more of them
                  tNat(VmRefCurrCurrency),tNat(VmRefCurBalMax)),
           local(tCommandType(VmInquiry2))):-
        getCommand(tCommandType(VmInquiry2),Tmp0),and(tBool(eq(tState(Vs0),tState(
            cx2p0(tState(cx2p0(tState(cx2p1(tState(cx2p0(tState(cone))))))))))),tBool(
            and(tBool(eq(Tmp0,tCommandCodeType(cLOADINIT))),tBool(VvDeactivated)))).
...
```

B.4 Test Oracle

```
oracle0(VPower_0_0,VCepCommand_3_0,VCepReply_4_0,VSelect_5_0,
        VSelectReply_6_0,VCepCommand_7_0,VCepReply_10_0,VCepReply_11_0,
        VCepCommand_12_0,VCepReply_29_0,VCepCommand_30_0,
        VCepReply_47_0,VCepCommand_48_0,VCepReply_65_0,
        VCepCommand_66_0,VCepReply_83_0,VCepCommand_84_0,
        VCepReply_85_0,VCepCommand_86_0,VCepReply_87_0,
        VCepCommand_88_0,VCepReply_89_0):-
        power(global(tState(cx2p0(tState(cx2p1(tState(cx2p0(tState(cx2p1(tState(cone))
            )))))))),
                  tNat(csucc(tNat(csucc(tNat(csucc(tNat(csucc(tNat(csucc(tNat(csucc
                      (tNat(csucc(tNat(csucc(tNat(csucc(tNat(csucc(tNat(csucc(tNat(
                      csucc(tNat(csucc(tNat(csucc(tNat(csucc(tNat(c0))))))))))))))))
                      ))))))))))))))),
                  ... % a lot more initialization parameters from
                      % the specification
                  tNat(c0), tNat(c0), tNat(csucc(tNat(csucc(tNat(csucc(tNat(csucc(
                      tNat(csucc(tNat(c0))))))))))),
            G1, local(VPower_0_0)),
        reset(G1, G2, local), aTR(G2, G3, local), cepCommand(G3, G4, local(
            VCepCommand_3_0)),
        ..., cepReply(G10, G11, local(VCepReply_10_0)), ...,
        cepCommand(G84, G85, local(VCepCommand_84_0)), cepReply(G85, G86, local(
            VCepReply_85_0)),
        cepCommand(G86, G87, local(VCepCommand_86_0)), cepReply(G87, G88, local(
            VCepReply_87_0)),
        cepCommand(G88, G89, local(VCepCommand_88_0)), cepReply(G89, _,   local(
            VCepReply_89_0)).
```

B.5 Oracle Query Results

```
[eclipse 2]: oracle(Power,CC1,CR1,S1,SR1,CC2,CR2,CR3,CC3,CR4,CC4,CR5,CC5,CR6,CC6,CR7,
    CC7,CR8,CC8,CR9,CC9,CR10).

Power = tPowerType(cON)
CC1 = tCommandType(ccommandData(tCommandCodeType(cLOADINIT), tNat(c0), tTxTypeType(
    cTxLoad), tNat(csucc(tNat(csucc(tNat(csucc(tNat(csucc(tNat(c0))))))))), tNat(csucc
    (tNat(csucc(tNat(csucc(tNat(csucc(tNat(csucc(tNat(c0))))))))))))
CR1 = tReplyType(creplyData(tNat(c0), tNat(c0), tNat(c0), tNat(c0), tNat(c0), tNat(c0)
    , tNat(c0), tNat(c0), tNat(c0), tNat(c0), tNat(c0), tNat(c0), tNat(c0), tNat(c0),
    tNat(c0), tStatusType(cx9101)))
S1 = tAidType(cCEP)
SR1 = tFCIType(cStatus(tStatusType(cx9000)))
... % solutions for CC2, CR2, CC3, CR3, CC4, CR4, CC5, CR5, CC6, CR6, CC7, CR7, CC8,
    CR8
CC9 = tCommandType(ccommandData(tCommandCodeType(cALLSLOTS01), tNat(V1), tTxTypeType(
    V2), tNat(V3), tNat(V4)))
CR10 = tReplyType(creplyData(tNat(csucc(tNat(csucc(tNat(csucc(tNat(c0)))))), tNat(
    csucc(tNat(csucc(tNat(csucc(tNat(c0)))))), tNat(csucc(tNat(csucc(tNat(csucc(tNat(
    c0)))))), tNat(csucc(tNat(csucc(tNat(csucc(tNat(c0)))))), tNat(c0), tNat(c0),
    tNat(c0), tNat(c0), tNat(c0), tNat(c0), tNat(c0), tNat(c0), tNat(c0), tNat(c0),
    tNat(c0), tStatusType(cx6A83)))
Yes (0.00s cpu, solution 1, maybe more) ? ;

... % second solution
Yes (0.00s cpu, solution 2, maybe more) ? ;

...
[eclipse 3]: _
```

References

1. Kent Beck. *Extreme Programming Explained*. Addison-Wesley, 1999.
2. S. C. C. Blom, 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, *13th International Conference on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 250–254. Springer-Verlag, 2001.
3. Jens R. Calamé. Specification-based Test Generation with TGV. Technical Report SEN-R0508, Centrum voor Wiskunde en Informatica, May 2005.
4. Jens R. Calamé, Natalia Ioustinova, Jaco van de Pol, and Natalia Sidorova. Data Abstraction and Constraint Solving for Conformance Testing. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 541–548. IEEE Press, 2005.
5. CEPSCO. *Common Electronic Purse Specifications, Functional Requirements*, September 1999. Version 6.3.
6. CEPSCO. *Common Electronic Purse Specifications, Technical Specification*, May 2000. Version 2.2.
7. Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Automated test and oracle generation for smart-card applications. In I. Attali and T. Jensen, editors, *E-smart 2001*, Lecture Notes in Computer Science, pages 58–70. Springer, 2001.
8. Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Stg: a tool for generating symbolic test programs and oracles from operational specifications. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 301–302, New York, NY, USA, 2001. ACM Press.
9. Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Stg: A symbolic test generation tool. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 470–475, London, UK, 2002. Springer-Verlag.
10. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM Press.

11. D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD dissertation, Eindhoven University of Technology, July 1996.
12. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2):253–291, 1997.
13. Hakan Erdogmus, Maurizio Morisio, and Marco Torchiano. On the effectiveness of the test-first approach on programming. *IEEE Transactions on Software Engineering*, 31(3):226–237, March 2005.
14. 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.
15. L. Frantzen, J. Tretmans, and T.A.C. Willemse. Test generation based on symbolic specifications. In J. Grabowski and B. Nielsen, editors, *FATES 2004*, number 3395 in LNCS, pages 1–15. Springer-Verlag, 2005.
16. Hubert Garavel and Frédéric Lang. Ntif: A general symbolic model for communicating sequential processes with data. In *Proc. of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'2002)*, 2002.
17. J. F. Groote, A. Ponse, and Y. S. Usenko. Linearization in parallel pCRL. *Journal of Logic and Algebraic Programming*, 48(1-2):39–72, 2001.
18. Jan Friso Groote and Alban Ponse. The syntax and semantics of μcrl. Technical report, Centrum voor Wiskunde en Informatica, 1990.
19. N. Ioustinova. *Abstractions and Static Analysis for Verifying Reactive Systems*. PhD thesis, Free University of Amsterdam, 2004.
20. N. Ioustinova, N. Sidorova, and M. Steffen. Closing open SDL-systems for model checking with DTSpin. In L. H. Eriksson and P. A. Lindsay, editors, *FME 2002: Formal Methods - Getting IT Right, Proc. of Int. Symposium of Formal Methods Europe, FME 2002*, volume 2391 of *Lecture Notes in Computer Science*, pages 531–548. Springer, 2002.
21. N. Ioustinova, N. Sidorova, and M. Steffen. Synchronous closing and flow abstraction for model checking timed systems. In *Proc. of the Second Int. Symposium on Formal Methods for Components and Objects (FMCO'03)*, volume 3188 of *Lecture Notes in Computer Science*. Springer, 2004.
22. C. Jard and T. Jéron. TGV: Theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 7(4):297–315, 2005.
23. B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva. Symbolic test selection based on approximate analysis. In *11th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)Volume 3440 of LNCS*, Edinburgh (Scottland), April 2005.
24. Jan Jürjens and Guido Wimmel. Formally testing fail-safety of electronic purse protocols. In *Automated Software Engineering (ASE 2001)*. IEEE Computer Society, 2001.
25. Jan Jürjens and Guido Wimmel. Security Modelling for Electronic Commerce: The Common Electronic Purse Specifications. In Beat Schmid, Katarina Stanoevska-Slabeva, and Volker Tschammer, editors, *Towards the E-Society. Proceedings of 1st IFIP International Conference on E-Commerce, E-Business and E-Government*, pages 489 – 506. Kluwer Academic Publishers, 2001.
26. Jan Jürjens. *Secure Systems Development with UML*. Springer, Berlin, Heidelberg, 2005.
27. Philippe Kruchten. Tutorial: introduction to the rational unified process. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 703–703, New York, NY, USA, 2002. ACM Press.
28. Kim Marriott and Peter J. Stuckey. *Programming with Constraints – An Introduction*. MIT Press, Cambridge, 1998.

29. Object Management Group. *MDA Guide*, 1.0.1 edition, June 2003.
30. A. Pretschner, O. Slotosch, E. Aiglstorfer, and S. Kriebel. Model-based testing for real. *International Journal on Software Tools for Technology Transfer*, 5(2–3):140–157, 2004. This article develops an approach for model-based test case generation whereby the abstract models are created by the TUM tool AutoFocus. The test case generation itself happens by symbolic execution using Constraint Logic Programming techniques. [48, 49].
31. V. Rusu, L. du Bousquet, and T. Jeron. An approach to symbolic test generation. In *Int. Conference on Integrating Formal Methods (IFM00)*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357, 2000.
32. Vlad Rusu. Verification using test generation techniques. In *FME '02: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, pages 252–271, London, UK, 2002. Springer-Verlag.
33. Natalia Sidorova and Martin Steffen. Embedding chaos. In Patrick Cousot, editor, *Proc. 8th International Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 319–334. Springer-Verlag, 2001.
34. J. Tretmans. Test generation with inputs, outputs, and repetitive quiescence. *Software - Concepts & Tools*, 17(3):103–120, 1996.
35. Guido Wimmel and Jan Jürjens. Specification-based Test Generation for Security-Critical Systems Using Mutations. In *ICFEM 2002, 2002 Shanghai, China*, pages 471 – 482. Springer-Verlag, October 2002.
36. ITU-T Recommendation X.290-ISO/IEC 9646-1, Information Technology- Open Systems Interconnection- Conformance Testing Methodology and Framework- Part 1: General Concepts.
37. Eléna Zinovieva-Leroux. *Méthodes symboliques pour la génération de tests de systèmes reactifs comportant des données.* PhD thesis, Université de Rennes, 2004.