Data Abstraction and Constraint Solving for Conformance Testing [*]

Jens R. Calamé, Natalia Ioustinova, Jaco van de Pol,
Centrum voor Wiskunde en Informatica, P.O. Box 94079
1090 GB Amsterdam, The Netherlands
Jens.Calame@cwi.nl Natalia.Ioustinova@cwi.nl
Jaco.van.de.Pol@cwi.nl

Natalia Sidorova
Eindhoven University of Technology
Den Dolech 2, P.O. Box 513
5612 MB Eindhoven, The Netherlands
n.sidorova@tue.nl

Abstract

Conformance testing is one of the most rigorous and well-developed testing techniques. Model-based test generation is an essential phase of the conformance testing approach. The main problem in this phase is the explosion of the number of test cases, often caused by large or infinite data domains for input and output data. In this paper we propose a test generation framework based on the use of data abstraction and constraint solving to suppress the number of test cases. The approach is evaluated on the CEPS (Common Electronic Purse Specifications) case study.

1. Introduction

Throughout the years, testing remains one of the most popular techniques that are used by industry to ensure the reliability of systems. The main purpose of testing is to discover as many defects in a system implementation as possible. A large number of testing techniques have been developed by academic and industrial communities to provide efficient and reliable ways of finding errors. *Conformance testing* [19] is one of the most rigorous among existing testing techniques. Given a specification, conformance testing is concerned with checking whether an implementation under test (IUT) conforms the specification.

Intuitively, an IUT conforms its specification if after each input foreseen by the specification, the IUT exhibits only the behavior allowed in the specification [17]. Assessing conformance of an IUT is done by executing *test cases*. A test case contains all possible reactions of an IUT on certain environmental inputs. Reactions of an IUT may lead to different verdicts: Verdict Fail denotes a violation of the specification while verdict Pass means that the reaction of an IUT is correct wrt. the specification. Checking all possible test cases is however not feasible, only a subset of system behavior can be tested. Therefore some *test selection* should be done prior to testing.

Using *test purposes* is one of the most popular strategies for selecting test cases. A test purpose defines a subset of the system behavior on which test cases should be focused. In case the IUT exhibits a behavior allowed by the specification but not fitting the test purpose, verdict Inconc (inconclusive) is given.

One way to generate test cases is to enumerate the state space of the specification and select a part of it satisfying a test purpose [11]. The generation does not always terminate and even if terminated it often produces a huge number of test cases. The most important reason the number explodes is that data exchanged within and with the system comes from large or even infinite domains. Here we propose a testing framework that alleviates this problem by combining data abstractions and constraint solving with enumerative test generation techniques.

We consider specifications that describe open systems communicating with their *environments*. Assumptions about the component's environment that software programmers are making is an important source of software errors. They are often not documented and in many cases erroneous. Two classical examples of failures caused by this kind of errors are an incorrect handling of an arithmetic exception that led to a power shutdown of cruiser USS Yorktown and an unanticipated floating-point exception that caused a rocket boost failure in Ariane 5 [16]. Therefore, for testing purposes, we consider the most general possible environment that can send all possible inputs wrt. the specification, parameterized with arbitrary data. Further we refer to these environments as *"chaotic"* environments.

Assuming a chaotic environment, we abstract the environmental data parameterizing inputs into one abstract value. The abstract system shows then at least the behavior of the original system [15]. We implement the abstrac-

[*] This work is done within the project "TTMedal. Test and Testing Methodologies for Advanced Languages (TT-Medal)" [18]

tion as a program transformation on the level of specification. Given a test purpose, we obtain an abstract test case by applying already existing enumerative test generation algorithms [11] to the abstract system that is derived from the transformed specification.

An abstract system is an overapproximation of the original one. Inputs and outputs of an abstract test case generated from the abstract system carry abstract values. To make the abstract test cases executable, abstract values should be concretized. We employ constraint solving to find concrete values that will substitute the abstract ones. Substituting all occurrences of abstract values is expensive and unnecessary. Therefore, we transform the original specification into a constraint logic program (CLP) and shift constraint solving to the test execution phase.

When testing a system, we want to reduce the number of inconclusive verdicts. For this purpose, we start test execution by choosing a shortest trace to verdict Pass. We transform the chosen trace into a query for the CLP. If the query has at least one solution, then a concretization of this abstract trace is present in the original specification. The solution provides concrete values to substitute occurrences of abstract values. In case there is no solution, the trace is introduced by the abstraction and has to be removed from the test case.

Having a substitution for just one trace, we start the test execution. If the IUT follows the selected trace all the way, verdict Pass is assigned. In case the reaction of the IUT deviates from the selected trace, there are two possibilities: the IUT violates the specification or it follows a different but still correct trace, which is possible due to nondeterminism in the specification. When the IUT violates the specification, we stop the test execution and assign the verdict Fail. Otherwise, we form a new query and a constraint solver is consulted again. If there is a solution then we proceed with the test execution. If no solution is found, verdict Inconc is assigned.

Here, we assume decidability of all guards in the specification, e.g. the guards might belong to a decidable fragment of Presburger arithmetic [14] with uninterpreted functions [1]. We implement our approach to generate test cases from μCRL specifications with TGV [11] and use the μCRL toolset [2] to specify systems and test purposes, and to generate and reduce the state space. Eclipse Prolog [7, 6] is used to implement constraint solving.

The rest of the paper is organized as follows: In Section 2, we give an overview of testing theory our approach is based on. Section 3 defines a set of specifications we work with and provides an implementation for the data abstraction. In Section 4, we first illustrate our approach to using the data abstraction for testing and then provide its generalization. We conclude with Section 5 where we also discuss related and future works.

2. Testing Theory

For the generation of abstract test cases from specifications, we rely on the approach to conformance test generation proposed in [11] and implemented in the tool TGV. This approach relates specifications with conforming implementations by a *conformance relation*. It formalizes the notions of a test case and a test purpose and also defines correctness criteria for test cases.

Specifications and implementations are modelled by input output labelled transition systems ($IOLTS$s). An $IOLTS\ M$ is given by a tuple $(\Sigma, Lab, \rightarrow_\lambda, \sigma_0)$, where $\Sigma \neq \emptyset$ is a set of states, Lab is a set of labels (*actions*), $\rightarrow_\lambda \subseteq \Sigma \times Lab \times \Sigma$ is a transition relation, and $\sigma_0 \in \Sigma$ is the initial state. The set of labels Lab consists of three subsets of actions, Lab_I, Lab_O, and $\{\tau\}$ denoting visible input, output and invisible internal actions. An $IOLTS$ is *deterministic* iff there is at most one outgoing transition for each action $\lambda \in Lab$ in each state $\sigma \in \Sigma$.

The behavior of an $IOLTS$ is given by sequences of states and transitions $\zeta = \sigma_0 \rightarrow_\lambda \sigma_1 \rightarrow_\lambda \ldots$ starting from the initial one. In *traces*, the states are projected out, i.e. $[\![M]\!]_{trace} \subseteq Lab^\star$, where $[\![M]\!]_{trace}$ denotes the set of traces of an $IOLTS\ M$. $IOLTS$s modelling IUTs are assumed to be *input-complete*, meaning, the implementation must accept any input from its environment.

Conformance testing is restricted to observing outputs (or deadlock) only after those traces that are contained in the specification. The specification maybe *partial*, in which case the output of the IUT after unspecified inputs is not restricted. The approach in [11] (following Tretmans in [17]) describes the set of conforming IUTs by an **ioco** relation on implementations and specifications. Given a model M_{IUT} of an implementation and a model M_{Spec} of a specification, the IUT is in **ioco**-relation with *Spec* if and only if for all traces β from M_{Spec}, whenever M_{IUT} can issue an output (or deadlock) after executing β, then also M_{Spec} can execute trace β followed by the same output (or deadlock). In this paper, we will not consider deadlocks.

The conformance test generation in [11] is guided by test purposes that are deterministic $IOLTS$s (denoted further M_{TP}) 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 sink states. Moreover, M_{TP} is complete in all the states except the accepting and refusing ones.

Test generation guided by a test purpose consists in building a standard *synchronous product* M_{SP} of M_{Spec} with M_{TP} and assigning verdicts. The Pass verdict is assigned to those states of the product which correspond to *accept* states in the test purpose. The Inconc verdict is assigned to the states from which accepting states are not reachable. The Fail verdict is implicit and is assigned after all unspecified outputs. Since the product represents expected behavior of an IUT from the tester's point of view, all input and output actions are mirrored during the generation of the product.

Test cases are derived from the product by resolving choices between several outputs and between inputs and outputs that might be present in the product. Formally, a *test case* M_{TC} is a deterministic input-complete $IOLTS$ equipped with sink states Pass, Inconc and Fail. A *test suite* is a collection of test cases.

Test execution consists in giving outputs of a test case as stimuli to an IUT and observing whether reactions of the IUT match inputs expected by the test case. Execution of test cases on an IUT should give a verdict about conformance of an IUT wrt. $Spec$. Verdicts assigned by a test case should be *sound* [11]. Fail is assigned if and only if a violation of the specification is observed. Pass is assigned if and only if we observe a trace that fits the test purpose and belongs to the specification. Inconc may be assigned only if an observed trace belongs to $Spec$ but is refused by the test purpose.

3. Data Abstraction

Inadequate and underspecified assumptions about environment are an important source of software errors. We aim at automatic generation of test cases that does not depend on particular assumptions about environmental behavior. Thus we use the most general, "chaotic", environment that can send and receive all possible messages in an arbitrary order. Values exchanged by the system with an environment are coming from large or even infinite domains. That immediately leads to large or infinite number of test case obtained during test case generation. Here we use a data abstraction that allows us to obtain an smaller (finite) overapproximation of the original system.

We abstract data coming from environment to one *"chaotic"* value, denoted by $\mathbb{T}$. Values that are not influenced by the environment remain unchanged. They should be treated in the same way as in the original system. This data abstraction was first proposed in [15] and successfully used for model checking open systems. A system obtained by this approach is a safe abstraction of the original one, meaning, it shows *at least* the behavior of the original system [15].

In this section, we first define the syntax and the semantics of specifications we work with and further explain the implementation of the data abstraction as a transformation on the level of specifications.

3.1. Syntax and Semantics

Our operational model is based on synchronously communicating processes with top-level concurrency. This is a simplification of a model used in [15]. A specification $Spec$ is given as the parallel composition $\Pi_{i=1}^{n} P_i$ of a finite number of processes. A process definition P is given by a four-tuple $(Var, Loc, \sigma_0, Edg)$, where Var denotes a finite set of variables, and Loc denotes a finite set of *locations*, or control states. A mapping of variables to values is called a valuation; we denote the set of valuations by $Val = \{\eta \mid \eta \colon Var \rightarrow D\}$. We assume standard data domains such as $\mathbb{N}$, $Bool$, etc. We write D when leaving the data-domain unspecified and silently assume all expressions to be well-typed. Let $\Sigma = Loc \times Val$ be the set of states, where a process has one designated initial state $\sigma_0 = (l_0, \eta_0) \in \Sigma$. The set $Edg \subseteq Loc \times Act \times Loc$ denotes the set of edges. An *edge* describes changes of configurations specified by an *action* from a set Act.

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 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 $\alpha, \alpha' \ldots$ 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 $\zeta = \sigma_0 \rightarrow_{\lambda} \sigma_1 \rightarrow_{\lambda} \ldots$ starting from the initial one. The step semantics is given by an $IOLTS\ M = (\Sigma, Lab, \rightarrow_{\lambda}, \sigma_0)$, where $\rightarrow_{\lambda} \subseteq \Sigma \times Lab \times \Sigma$ is given as a labelled transition relation between states. The labels differentiate between internal τ-steps and communication steps, either input or output, which are labelled by a signal and a value being transmitted, i.e. $?s(v)$ or $!s(v)$, respectively.

Receiving a signal s with a communication parameter x, $l \longrightarrow_{?s(x)} \hat{l} \in Edg$, results in updating the valuation $\eta_{[x \mapsto v]}$ according to the parameter of the signal and changing current location to $\hat{l}$. 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 the change of the location of the process from l to $\hat{l}$.

Assignments, $l \longrightarrow_{g \rhd x:=e} \hat{l} \in Edg$, result in the change of a location and the update of the valuation $\eta_{[x \mapsto v]}$, where $[\![e]\!]_{\eta} = v$. Assignments are internal, so assignment transitions are labelled by τ.

3.2. Program Transformation

Abstraction theory is well developed within the Abstract Interpretation framework [4, 5]. Here we provide a variation of the program transformation from [9] implementing the data abstraction.

We extend each data domain by an additional value $\mathbb{T}$, i.e. we assume abstract domains $\mathbb{N}^{\mathbb{T}} = \mathbb{N} \cup \{\mathbb{T}_{\mathbb{N}}\}$, $Bool^{\mathbb{T}} = Bool \cup \{\mathbb{T}_{Bool}\}$ etc. These $\mathbb{T}$-values are considered as the largest ones.

The transformation of the process specification consists in lifting all variables, expressions and guards to $\mathbb{T}$-data domains. Each occurrence of a variable x carrying values from domain D, is substituted by an occurrence of the variable $x^{\mathbb{T}}$ carrying values of domain $D^{\mathbb{T}}$.

Each expression e is strictly lifted to expression $e^{\mathbb{T}}$. If at least one of the variables of $e^{\mathbb{T}}$ carries a $\mathbb{T}$-value, then $[\![e^{\mathbb{T}}]\!]_{\eta^{\mathbb{T}}} = \mathbb{T}$. Otherwise, expression $e^{\mathbb{T}}$ has the same value as in the original system.

The transformation of guards is similar to the transformation of expressions. Every occurrence of a guard g is lifted to a guard $g^{\mathbb{T}}$ of type $Bool^{\mathbb{T}}$. While transforming guards we should ensure that the abstract system shows *at least* the behavior of the original system. The guards valuated to $\mathbb{T}$ behave as guards valuated to $true$. To avoid introducing unnecessary nondeterminism, we provide a more refined lifting for boolean operations.

After lifting system variables, expressions and guards, we obtain a system that can receive all values defined by the original specification as well as $\mathbb{T}$-values from the environment. The environment can influence data only via inputs. We transform every input $l \longrightarrow_{?s(x)} \hat{l}$ from the environment into an input of signal s parameterized by the $\mathbb{T}$-value from the corresponding domain followed by assigning this $\mathbb{T}$-value to the variable $x^{\mathbb{T}}$, i.e. every input edge is substituted by $l \longrightarrow_{?s(\mathbb{T})} \longrightarrow_{true \, \triangleright \, x^{\mathbb{T}} := \mathbb{T}} \hat{l} \in Edg^{\mathbb{T}}$.

A specification obtained by this transformation is referred further as $Spec^{\mathbb{T}}$. An abstract system modelling the transformed specification is referred further as $M^{\mathbb{T}}$. This system can receive only $\mathbb{T}$ values from the environment, so the infinity of the environmental data is collapsed into one value. Basically, the transformed system shows at least the traces of the original system where data influenced by environment are substituted by $\mathbb{T}$ values [10].

Although we provide here the program transformation for specifications consisting of one process only, it does not limit our approach. Existing linearization techniques [8] allow to obtain a single process definition for a parallel composition of a finite number of process definitions by resolving communication and parallel composition.

4. Testing with abstraction

In this section, we first illustrate our approach on a simple example and then generalize it.

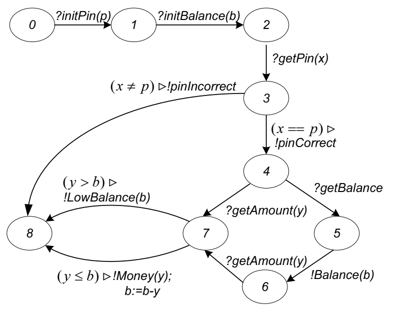

Figure 1. Simple Cash Machine

4.1. Cash Machine Example

Fig. 1 gives a specification of a simple cash machine. When a customer enters a card, the machine initializes a pin and a balance according to the card. Then the customer should enter a pin number. The machine checks whether the entered pin is the same as the one it expects after the initialization. If an incorrect pin is entered, the machine sends the $pinIncorrect$-message and stops the communication with the customer. Otherwise, it sends the $pinCorrect$-message and allows the user to choose between two options: (i) to withdraw some cash or (ii) to ask information about the current balance and then to get money. In case the customer tries to withdraw more money than the balance, the message $LowBalance$ parameterized by the balance value is issued. Otherwise, the machine dispenses money to the customer and updates the balance.

As a *test purpose* we choose entering a pin followed (after some steps) by dispensing some money. Already for this simple specification, test generation yields a large number of test cases, which is caused by large domains for input values. Therefore we produce an abstract specification by transforming the original one as described in Section 3. The simplified result of the transformation is given in Fig. 2. Then we apply the enumerative test generation [11] to obtain the synchronous product of the transformed specification and the test purpose. The result of the generation is given in Fig. 3.

To present the (expected) behavior of the system from the tester point of view, inputs and outputs in the product are mirrored with the specification. The product also contains verdicts. Note that the product might implement multiple test strategies - after entering a pin we still can choose between asking for some cash and inquiring the balance. Since we want testing results to be repeatable, we choose one of the options, namely, withdrawing money without asking for the balance. The selected test case is depicted in solid lines.

To make the selected test case executable, we need to concretize abstract values in it. We employ constraint solving to find concrete values. The information about con-

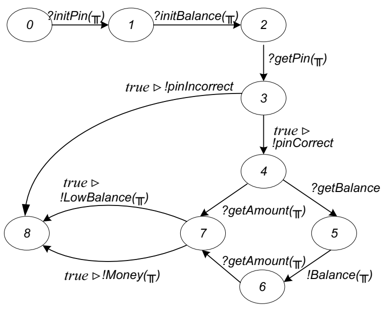

Figure 2. Transformed Cash Machine

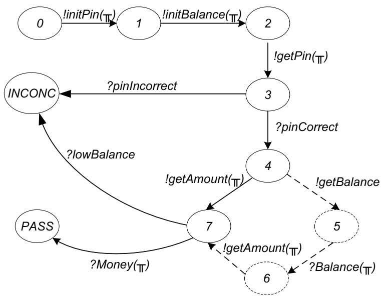

Figure 3. Synchronous product with verdicts

crete data and their dependencies is contained in the original specification. Therefore, we transform the specification into a constraint logic program. The fragment of the program for the edges $getPin$, $pinIncorrect$ and $pinCorrect$ is given in Fig. 4.

The constraint logic program is a set of rules that consist of an action name, three parameters and a guard that can be empty. The first parameter given by the first $state$-tuple represents states in which the action is enabled. The second parameter given by the second $state$-tuple represents states reachable by the action. The third parameter, the $param$-tuple, contains variables local to the rule, which are introduced to represent input and output data.

In the fragment in Fig. 4, variables P, B, X, Y of the specification represent respectively the initialized pin, the initialized balance, the pin entered by the customer, and the required money. In the rule $getPin(state(2, P, B, X, Y),$ $state(3, P, B, X_1, Y),$ $param(X_1))$, variable X_1 represents the value entered by the customer and $getPin$-action substitutes the value of the variable X of the original specification by the value of X_1. This action is possible only in location 2 and it leads to location 3. The other variables remain unchanged.

We are interested in getting Pass or Fail verdicts and want to avoid Inconc verdicts because test executions ending with Inconc verdict neither cover the behavior of interest nor discover software failures. Therefore, we choose a

% Constraint logic program ...
$getPin(state(2, P, B, X, Y), state(3, P, B, X_1, Y),$
  $param(X_1)).$
$pinIncorrect(state(3, P, B, X, Y), state(8, P, B, X, Y),$
  $param(\_)) :- X \neq Pin.$
$pinCorrect(state(3, P, B, X, Y), state(4, P, B, X, Y),$
  $param(\_)) :- X = Pin.$
...
% Query
$oracle(P, B, X, Yin, Yout) :-$
  $initPin(state(0, 0, 0, 0, 0), G_1, param(P)),$
  $initBalance(G_1, G_2, param(B)),$
  $getPin(G_2, G_3, param(X)),$
  $pinCorrect(G_3, G_4, \_),$
  $getAmount(G_4, G_5, param(Yin)),$
  $money(G_5, \_, param(Yout)).$

Figure 4. Query and a fragment of the constraint logic program

$[eclipse2] : oracle(P, B, X, Yin, Yout).$
$P = P\{-1.0inf..1.0inf\}$ $B = B\{-1.0inf..1.0inf\}$
$X = P\{-1.0inf..1.0inf\}$ $Yin = Yout\{-1.0inf..1.0inf\}$
$Yout = Yout\{-1.0inf..1.0inf\}$
Delayed goals:
$Yout\{-1.0inf..1.0inf\} - B\{-1.0inf..1.0inf\} =< 0$
Yes

Figure 5. Solver's output

trace of the abstract test case to Pass and transform it into a query for the constraint logic program. The query for the Pass-trace of the abstract test case is given in Fig. 4. In the query, variables P, B, X, Yin, $Yout$ denote respectively the initial pin, the initial balance, the pin entered by a customer, the amount of money requested by the customer and the amount of money dispensed by the cash machine. We use *Eclipse Prolog* [7, 6] to solve the query.

The output of the solver is given in Fig. 5. The query has solutions and the solver returns the information about them: The pin-value entered by the customer should be the same as the value of the initialized pin and the amount of money dispensed by the machine should be the same as the amount required by the user. The solver also provides an additional information, delayed goals, meaning the amount dispensed by the machine should not be greater than the initial balance.

Assume we have chosen a substitution $\{P \mapsto 888, B \mapsto 9999, X \mapsto 888, Yin \mapsto 9000, Yout \mapsto 9000\}$ that is one of the possible solutions of the query. We start the test execution that consists in providing stimuli to an IUT and checking whether the response of the IUT is the same as we expect according to the selected trace and data.

Assume that during the execution of selected trace we observe a reaction of the IUT that does not match the expected one according to the selected trace. For example, after sending request $getAmount(9000)$ to the IUT, it returns $Money(9001)$ instead of $Money(9000)$. For the cash machine, this reaction is obviously a wrong one because the substitution $\{P \mapsto 888, B \mapsto 9999, X \mapsto 888, Yin \mapsto 9000, Yout \mapsto 9001\}$ is not a solution for the query. Therefore, we issue verdict Fail and stop the test execution.

Specifications might be nondeterministic, so more than one correct response is possible on one input. In this case, assigning a verdict or proceeding test execution will require an analysis of responses obtained from the IUT, which we explain in the next subsection.

4.2. The Approach

Here we generalize our testing framework that covers the *test selection* and *test execution* phases.

Test selection consists of the following steps: (1) Given a test purpose TP and a specification Spec, we transform the specification into the abstracted one, $\mathit{Spec}^{\mathbb{T}}$. (2) Then we apply the enumerative test generation algorithm from [11] to obtain a synchronous product of $IOLTS$s modelling $\mathit{Spec}^{\mathbb{T}}$ and TP. (3) Abstract test cases are selected from the product by resolving choices between several outputs or between inputs and outputs that might exist in some states of the product [11]. (4) To execute an abstract test case, abstract data should be concretized. Prior to executing the test case, we concretize only a part of abstract values sufficient to start the execution. Having a partial concretization we start *executing* the test case, while the concretization of other abstract values happens "on-demand", i.e. it is shifted to the test execution phase. Now we consider these steps in more detail.

The transformation of a specification into an abstract one is given in Section 3. The product generated from the abstract specification and the test purpose might implement many testing strategies. To keep testing results repeatable, we have to select one of the strategies. By pruning conflicting inputs and outputs [11], we single-out an input-complete subgraph of the product. It does not contain choices between several outputs or choices between inputs and outputs. We further refer to the subgraph as an *abstract test case*. Here, we limit our attention to subgraphs without loops.

Environmental data occurring in the product are abstracted into $\mathbb{T}$-values. To make the abstract test case executable, abstract values should be concretized. For the concretization of abstract values, we employ constraint solving [13]. The information necessary for concretization is

$$\frac{l \longrightarrow_{g \,\triangleright !s(e)} \hat{l} \in Edg}{s(state(l,\tilde{\eta}), state(\hat{l},\tilde{\eta}), param(X)) :- \, g \wedge X = e}\ \textsc{ROut}$$

$$\frac{l \longrightarrow_{?s(x)} \hat{l} \in Edg}{s(state(l,\tilde{\eta}), state(\hat{l},\tilde{\eta}_{[x \mapsto Y]}), param(Y))}\ \textsc{RInp}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, x:=e} \hat{l} \in Edg}{\tau(state(l,\tilde{\eta}), state(\hat{l},\tilde{\eta}_{[x \mapsto e]}), param(\_)) :- \, g}\ \textsc{RAssign}$$

Table 1. From specification Spec to constraint logic program $\mathcal{RS}$

contained in the original specification. Therefore, we transform it into a constraint logic program $\mathcal{RS}$ that consists of rules. Each edge of the specification is mapped into a rule as defined in Table 1.

Rules of the constraint program are of the following form: $name(state(l,\tilde{\eta}), state(\hat{l},\tilde{\eta}'), param(Y)) :- g$ where $name(state(l,\tilde{\eta}), state(\hat{l},\tilde{\eta}'), param(Y))$ is a *user defined* constraint and g is a guard. The first parameter $state$ of the constraint describes the source states in terms of locations and valuations of process variables. The second parameter $state$ describes the destination states in terms of locations and valuations of process variables. The third parameter $param$ contains parameters representing input and output values. The constraint is satisfied iff the guard g is satisfied.

By ROUT, the output edge $l \longrightarrow_{g \,\triangleright !s(e)} \hat{l}$ is transformed into the rule $s(state(l,\tilde{\eta}),\ state(\hat{l},\tilde{\eta}),\ param(X)) :- (g \wedge X = e)$. The name of the constraint coincides with signal s. The edge leads to the change of location from l to $\hat{l}$. The values of the process variables $\tilde{\eta}$ remain unmodified. The output value is represented by parameter X. The value of this variable is given by expression e.

By RINPUT, the input edge $l \longrightarrow_{?s(x)} \hat{l}$ is transformed into the rule $s(state(l,\tilde{\eta}),\ state(\hat{l},\tilde{\eta}_{[x \mapsto Y]}),\ param(Y))$. Here, input leads to the substitution of process variable x by input parameter Y.

By RASSIGN, an assign-edge $l \longrightarrow_{g \,\triangleright\, x:=e} \hat{l}$ is mapped into a τ-rule $\tau(state(l,\tilde{\eta}),\ state(\hat{l},\tilde{\eta}_{[x \mapsto e]}),\ param(\_)) :- g$. An assignment is represented by substituting process variable x by expression e. τ-rules have no local parameters, which is denoted by the underscore (*don't-care*) here.

Specifications are often nondeterministic meaning multiple reactions might be specified for one input. We cannot predict which of the reactions are implemented by an IUT, so concretizing all abstract values is time-consuming and unnecessary. Therefore, we try to find a concretization only for abstract values of one trace and shift the concretization of other abstract values to the execution phase.

Different occurrences of the same abstract value do not necessarily represent the same concrete value. In order to differentiate the occurrences of abstract values, we substitute each occurrence of $\mathbb{T}$ values by a unique variable that does not occur in the original specification. We will further refer to these variables as *parameters* of the abstract test case.

Formally, an abstract test case is an input complete *IOLTS* $M^{\mathbb{T}}_{TC}$ equipped with a set of parameters Var_{param}. Test case $M^{\mathbb{T}}_{TC}$ might contain several Pass-traces. We select one (for instance, the shortest one) Pass-trace β of $M^{\mathbb{T}}_{TC}$ and transform it into a query $\mathcal{O}_{\beta}$ for the constraint logic program.

Basically, a query is a conjunction of constraints corresponding to the steps of the selected Pass-trace β. We construct the query by induction on the length of the selected trace. Initially, the query is empty and the first step of the trace becomes the first element of the conjunction.

The initial input(output)-step, $\sigma_{init} \longrightarrow_{?s(Y)} \hat{\sigma}$ ($\sigma \longrightarrow_{!s(Y)} \hat{\sigma}$), is transformed into a query $s(state(l_{init}, \tilde{\eta}_{init}),\ state(L_1, \tilde{\eta}_1),\ param(Y))$. There l_{init} is the initial location of $Spec$, variable L_1 denotes the location of $Spec$ reachable by the step. $\tilde{\eta}_{init}$ gives the initial valuation of system variables, $\tilde{\eta}_1$ denotes a valuation of process variables reachable from the initial one by the step. Parameter Y represents an input (output) value. The initial τ-step, $\sigma_{init} \rightarrow_{\tau} \hat{\sigma}$, is mapped into the query $\tau(state(l_{init}\ , \tilde{\eta}),\ state(L_1, \tilde{\eta}_1), param(\_))$.

The next input(output)-step $\sigma \longrightarrow_{?s(Y)} \hat{\sigma}$ ($\sigma \longrightarrow_{!s(Y)} \hat{\sigma}$) is transformed into the constraint $s(state(L_{(k+1)},\ \tilde{\eta}_{(k+1)}),\ state(L_{(k+2)},\ \tilde{\eta}_{(k+2)}),\ param(Y))$. The next τ-step $\sigma \rightarrow_{\tau} \hat{\sigma}$ is mapped into the constraint $\tau(state(L_{(k+1)},\ \tilde{\eta}_{(k+1)}),\ state(L_{(k+2)},\ \tilde{\eta}_{(k+2)}),\ param(\_))$. The constraint is added to the already constructed query by conjunction.

We use *Eclipse Prolog* [7, 6] to solve the query. If there is no solution for the query, the selected trace is introduced by the data abstraction. Therefore, we remove the trace from the test case. If none of the Pass traces of the abstract test case is solvable, we proceed by selecting another abstract test case from the product. If there is at least one solution for the query, the trace β can be mapped to a trace of the original system.

Let $\theta : Var_{param} \rightarrow D$ be a solution of the query in the rule system $\mathcal{RS}$. We refer to trace β with parameters substituted according to θ as an *instantiated trace*, denoted $[\![\beta]\!]_{\theta}$. By the construction of the constraint logic program and the query, the instantiated trace $[\![\beta]\!]_{\theta}$ is a trace of the original system.

Test execution Knowing one possible solution for the selected Pass-trace, we start the execution of the abstract test case $M^{\mathbb{T}}_{TC}$. Test execution consists in giving output-steps as stimuli to an IUT and checking whether reactions of the IUT match inputs expected according to the selected trace. The τ-steps are not observable, so we just skip them during the execution. In case responses match expected inputs, we just proceed the execution until the Pass verdict is reached.

An IUT does not always follow the selected trace during the test execution: The IUT might provide an input $?s'(v')$ that does not match the input expected according to the selected trace. In this case, we first need to decide whether this input violates the specification. Let ρ be the already executed prefix of $[\![\beta]\!]_{\theta}$. We transform ρ followed by the observed input $?s'(v')$ into the new query for the constraint system. If the query has no solution, meaning, the observed input violates the specification, we assign the Fail verdict and stop the test execution. If the query has a solution, then the observed input does not violate the specification and we may proceed the test execution.

To proceed, we first check whether the abstract test case has a trace to Pass with prefix $\rho?s'(v')$. If there is such a trace, we transform ρ followed first by $?s'(v')$ and then by the selected trace to Pass to a new query. If the query has a solution, we use the solution to proceed with the test execution. Otherwise, we stop the test execution. We also cannot proceed if there is no trace to Pass after ρ followed by $?s'(v')$. Since we do not observe any violation of the specification in both cases, we assign verdict Inconc.

The abstract system shows at least the behavior of the original one and assigning test case verdicts is based on solving queries for the rule system obtained from the original specification. Therefore the test case verdicts assigned in result of the proposed test execution are *sound*.

Implementation and CEPS case study We have implemented our approach for test generation from μCRL specifications (see *www.cwi.nl/˜calame/dataabstr.html* for more detail). We use the μCRL toolset [2] to specify systems and test purposes, to generate and to optimize state spaces. TGV [11] is employed to generate abstract test cases from abstract systems. *Eclipse Prolog* [7, 6] is used to solve queries.

We evaluated our approach to test generation on **C**ommon **E**lectronic **P**urse **S**pecifications (CEPS) [3]. CEPS define a protocol for electronic payment using a multi-currency smart-card. A card has a number of slots, each corresponds to one currency and the balance for this currency. CEPS defines how the information stored in slots can be loaded, accessed and modified.

We have specified inquiry and load functionality of CEPS and performed test case generation for the test purpose based on the *load transaction* processing. We applied our abstraction tool to the specification and then instantiated and reduced the abstracted specification using the μCRL toolset. The instantiation and reduction took 16 minutes 5 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). Using TGV, we generated two test cases without loops: one of 594 states with 597 transitions and another one of 109 states with 111 transitions. Test case generation took 0.65 seconds and 0.42 seconds, respectively, 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 more detail see $www.cwi.nl/\tilde{}calame/dataabstr.html$.

5. Conclusion

In this paper we proposed an approach to test generation combining data abstraction and constraint solving with enumerative test generation techniques. Application of the approach to the CEPS case study shows that it is scalable to systems of industrial size. Currently we are working on the automation of the test execution process as described in Section 4.

The closest to our approach is *symbolic test generation* [12]. 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. Since coreachability problem is undecidable for the symbolic case, the coreachability analysis is overapproximated by classical Abstract Interpretation technique [4].

The purpose and usage of abstraction techniques in our approach is conceptually different from the one of symbolic test generation. We use a data abstraction that mitigates infinity of external data. It allows to obtain abstract test cases by applying existing enumerative test generation algorithms. Abstract test cases are further supplied by concrete data derived by constraint solving. In the symbolic test generation approach, approximate coreachability analysis is used for pruning pathes potentially not leading to Pass-verdicts. Both approaches are valid for any abstraction leading to an overapproximation of system behaviors. Both approaches employ constraint solving to choose a single testing strategy during test execution. More case studies are still needed to draw conclusions which approach is more suitable for which class of systems.

References

[1] W. Ackermann. *Solvable Cases of the Decision Problem*. North-Holland Publishing Company, 1954.

[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, *Proceedings 13th Conference on Computer Aided Verification (CAV'01), Paris, France*, volume 2102 of *Lecture Notes in Computer Science*, pages 250–254. Springer-Verlag, 2001.

[3] CEPSCO. *Common Electronic Purse Specifications, Technical Specification*, May 2000. Version 2.2.

[4] P. Cousot and R. Cousot. Abstract Interpretaion: 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*. ACM, January 1977.

[5] D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD dissertation, Eindhoven University of Technology, July 1996.

[6] *ECLIPSe Constraint Library Manual*, version 5.8 edition.

[7] *ECLIPSe User Manual*, version 5.8 edition.

[8] 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.

[9] N. Ioustinova, N. Sidorova, and M. Steffen. Abstraction and flow analysis for model checking open asynchronous systems. In *Proc. of the 9th Asia Pacific Software Engineering Conference (APSEC 2002)*, pages 227–235. IEEE Computer Society, 2002.

[10] 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.

[11] C. Jard and T. Jéron. TGV: theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Software Tools for Technology Transfer (STTT)*, 6, October 2004.

[12] 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.

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

[14] M. Presburger. Über die vollständigkeit eines gewissen systems der arithmetik ganzer zahlen in welchem die addition als einzige operation hervortritt. In *Sprawozdanie z I Kongress Matematykow Krajow Slowacanskich Warszawa*, pages 92–101, 1929.

[15] N. Sidorova and M. Steffen. Embedding chaos. In P. Cousot, editor, *Proc. of the 8th Static Analysis Symposium (SAS'01)*, volume 2126 of *Lecture Notes in Computer Science*, pages 319–334. Springer-Verlag, 2001.

[16] G. E. Thaller. *Software-Test*. Verlag Heinz Heise, 2000.

[17] J. Tretmans. Test generation with inputs, outputs, and repetitive quiescence. *Software - Concepts & Tools*, 17, 1996.

[18] TTMedal. Testing and Testing Methodologies for Advanced Languages. http://www.tt-medal.org.

[19] ITU-T Recommendation X.290-ISO/IEC 9646-1, Information Technology- Open Systems Interconnection- Conformance Testing Methodology and Framework- Part 1: General Concepts.