On-the-Fly Conformance Testing Using Spin

René de Vries and Jan Tretmans *
University of Twente
Formal Methods and Tools group, Department of Computer Science
P.O. Box 217, 7500 AE Enschede, The Netherlands
{rdevries,tretmans}@cs.utwente.nl

Abstract

In this paper we report about the construction of a tool for conformance testing based on Spin. The Spin tool has been adapted such it can derive test primitives from systems described in Promela. These primitives support the on-the-fly conformance testing process.

Traditional derivation of tests from formal specifications suffers from the state-space explosion problem and from complexity. Spin is one of the most advanced model checkers with respect to handling large state spaces. This advantage of Spin has been used for the derivation of test primitives from a Promela description.

To reduce the state space, we introduce the on-the-fly testing framework. Within this framework the Primer is distinguished. This Primer derives test primitives from a model of a system according to a well defined and complete testing theory. Algorithms are presented which enable us to derive test primitives from a Promela description. These algorithms have been implemented in the adapted version of the Spin tool which acts as the Primer in the framework. As a result of this prototype study it is concluded that it is in principle possible to derive these primitives automatically from Promela descriptions, and to perform testing.

1 Introduction

Testing is the activity of doing experiments with system implementations in order to gain confidence in their correct functioning. What correct functioning is, is determined by the system specification, which captures its functional behaviour. Preferably, this specification is given using a formal language, e.g., Lotos, Estelle, SDL, Z or Promela. Such formal languages have well defined semantics and do not suffer from problems of ambiguity and impreciseness, thus making them suitable as the basis for validation, implementation and testing. Moreover, formal languages allow processing by tools.

Whereas formal system verification is aimed at checking properties of a system by exercising a model of it, testing is aimed at exercising the real, physical system. Due to complexity inherent in most systems, testing can only exercise part of all possible system behaviour and, consequently, can never lead to certainty about the satisfaction of a property. Using model checking, on the other hand, system properties can be proved (with more or less rigour), however, this prove only applies to the model of the system and not to the real, physical system.

An example of a formal verification tool is the model checker Spin [Hol91]. This tool can be used to support system validation and verification by automatically assessing the validity of a property expressed in Linear Temporal Logic (LTL). This assesment is performed on a system

*This research was partly supported by the Dutch Technology Foundation STW under project STW TIF.4111: *Côte de Resyste* – COnformance TEsting of REactive SYSTEms.

model expressed in the formal language PROMELA. A PROMELA model of the system design or implementation is developed especially for checking such properties.

Once a PROMELA system model is available, one might consider to (ab)use this model as the basis for the generation of tests to test system implementations. In this way, the PROMELA model is regarded as the specification of prescribed behaviour. A next step then is to develop tools to automate the derivation of tests from system descriptions in PROMELA. This paper describes the development of such a tool.

In order to derive test cases, we need, apart from a specification, a test derivation algorithm. Moreover, to express such an algorithm and to reason about it (its soundness and exhaustiveness) we need to express formally when an implementation conforms to a specification. This is done by defining an *implementation relation* between the class of envisaged implementations and the class of specifications [ISO96]. In the realm of PROMELA, specifications and implementations can both be conceived as special kinds of *labelled transition systems*. Implementations are modelled as *input-output transition systems*, a kind of transition system where inputs are always enabled. Specifications are labelled transition systems in which inputs and outputs can be distinguished (but not necessarily always enabled). Hence, an implementation relation in this realm is a relation between input-output transition systems and labelled transition systems. We take the relation **ioco** introduced in [Tre96] together with the corresponding test derivation algorithm as our theoretical basis. For a rationale for this particular implementation relation we refer to [Tre96].

The goal of this paper is to report about the construction of a tool for conformance testing based on PROMELA specifications and the implementation relation **ioco**. The tool derives test cases and also to immediately execute them, i.e., performs *on-the-fly* test derivation and execution. The tool has been implemented based on SPIN, adapting SPIN to generate the information needed in the test derivation algorithm, and taking advantage of the capabilities of SPIN in dealing with large state spaces.

We start in the next section with recalling the formal models, the implementation relation **ioco** and the test derivation algorithm from [Tre96]. In Section 3 we elaborate on the on-the-fly method of testing and we discuss the tool architecture for test derivation and execution. Section 4 explains how the test derivation algorithm and on-the-fly testing can be applied to PROMELA and SPIN, what restrictions and assumptions are necessary for PROMELA descriptions to be viewed as transition system specifications, and how advantage can be taken from SPIN as the basis for the implementation of the test derivation tool. Section 5 explains what has been achieved, what lessons were learned and what remains to be done.

2 Formal Preliminaries

This section recalls those aspects of [Tre96] which are used to develop the test derivation algorithm and tool for PROMELA.

Labelled transition systems Labelled transition systems provide a formalism to specify, model, analyse and reason about system behaviour. A labelled transition system description is defined in terms of states and labelled transitions between states.

Definition 2.1
A *labelled transition system* is a 4-tuple $\langle S, L, T, s_0 \rangle$ where

- S is a non-empty set of *states*;
- L is a finite set of *labels*;
- $T \subseteq S \times (L \cup \{\tau\}) \times S$ is a set of triples, the *transition relation*;

- $s_0 \in S$ is the *initial state*.

$\square$

The labels in L represent the observable interactions of a system. The special label $\tau \notin L$ represents an unobservable, internal action. We denote the class of all labelled transition systems over L by $\mathcal{LTS}(L)$. Transition systems without infinite compositions of transitions with only internal actions are called *strongly converging*. For technical reasons we restrict $\mathcal{LTS}(L)$ to strongly converging transition systems.

A *trace* is a finite sequence of observable actions. The set of all traces over L is denoted by L^*, with ϵ denoting the empty sequence. If $\sigma_1, \sigma_2 \in L^*$, then $\sigma_1{\cdot}\sigma_2$ is the concatenation of σ_1 and σ_2. Some additional notations and properties are introduced in definitions 2.2 and 2.3.

Definition 2.2
Let $p = \langle S, L, T, s_0 \rangle$ be a labelled transition system with $s, s' \in S$, and let $\mu_i \in L \cup \{\tau\}$, $a_i \in L$, and $\sigma \in L^*$.

$$\begin{array}{lll}
s \xrightarrow{\epsilon} s' & =_{\text{def}} & s = s' \\
s \xrightarrow{\mu} s' & =_{\text{def}} & (s, \mu, s') \in T \\
s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' & =_{\text{def}} & \exists s_0, \ldots, s_n :\ s = s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} s_n = s' \\
s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} & =_{\text{def}} & \exists s' :\ s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' \\
s \not\xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} & =_{\text{def}} & \text{not } \exists s' :\ s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' \\
s \overset{\epsilon}{\Longrightarrow} s' & =_{\text{def}} & s = s' \ \text{ or } \ s \xrightarrow{\tau \cdot \ldots \cdot \tau} s' \\
s \overset{a}{\Longrightarrow} s' & =_{\text{def}} & \exists s_1, s_2 :\ s \overset{\epsilon}{\Longrightarrow} s_1 \xrightarrow{a} s_2 \overset{\epsilon}{\Longrightarrow} s' \\
s \overset{a_1 \cdot \ldots \cdot a_n}{\Longrightarrow} s' & =_{\text{def}} & \exists s_0 \ldots s_n :\ s = s_0 \overset{a_1}{\Longrightarrow} s_1 \overset{a_2}{\Longrightarrow} \ldots \overset{a_n}{\Longrightarrow} s_n = s' \\
s \overset{\sigma}{\Longrightarrow} & =_{\text{def}} & \exists s' :\ s \overset{\sigma}{\Longrightarrow} s' \\
s \overset{\sigma}{\not\Longrightarrow} & =_{\text{def}} & \text{not } \exists s' : s \overset{\sigma}{\Longrightarrow} s'
\end{array}$$

$\square$

We will not always distinguish between a labelled transition system and its initial state: if $p = \langle S, L, T, s_0 \rangle$, then we will identify the process p with its initial state s_0, and we write, for example, $p \overset{\sigma}{\Longrightarrow}$ instead of $s_0 \overset{\sigma}{\Longrightarrow}$.

Definition 2.3
Let p be a (state of a) labelled transition system and let P a set of states.

1. $\mathit{init}(p) \ =_{\text{def}} \ \{\ \mu \in L \cup \{\tau\} \mid p \xrightarrow{\mu}\ \}$
2. $\mathit{init}(P) \ =_{\text{def}} \ \bigcup \{\ \mathit{init}(p) \mid p \in P\ \}$
3. $\mathit{traces}(p) \ =_{\text{def}} \ \{\ \sigma \in L^* \mid p \overset{\sigma}{\Longrightarrow}\ \}$
4. $p\ \mathbf{after}\ \sigma \ =_{\text{def}} \ \{\ p' \mid p \overset{\sigma}{\Longrightarrow} p'\ \}$
5. $P\ \mathbf{after}\ \sigma \ =_{\text{def}} \ \bigcup \{\ p\ \mathbf{after}\ \sigma \mid p \in P\ \}$

$\square$

Input-output transition systems We assume that the label set L can be partitioned into input actions L_I and output actions L_U: $L = L_I \cup L_U$ and $L_I \cap L_U = \emptyset$. Moreover, we consider systems which always accept any input. In terms of transition systems: all inputs, i.e., all actions in L_I, are always enabled in any state of the transition system. Such transition systems are called *input-output transition systems*. In input-output transition systems, inputs of one system communicate with the outputs of the other system, and vice versa, (cf. IOA [LT89]).

Definition 2.4
An *input-output transition system* p is a labelled transition system in which the set of actions L is partitioned into input actions L_I and output actions L_U ($L_I \cup L_U = L, L_I \cap L_U = \emptyset$), and for which all inputs are always enabled in any state:

$$\text{whenever} \quad p \overset{\sigma}{\Longrightarrow} p' \quad \text{then} \quad \forall a \in L_I :\ p' \overset{a}{\Longrightarrow}$$

The class of input-output transition systems with input actions in L_I and output actions in L_U is denoted by $\mathcal{IOTS}(L_I, L_U) \subseteq \mathcal{LTS}(L_I \cup L_U)$.

□

Implementation relation The major issue of conformance testing is to decide whether an implementation is correct with respect to a specification. This requires a notion of correctness, which is covered by defining an *implementation relation*. An implementation relation is a (formal) relation between the domain of specifications and the domain of models of implementations, such that (i, s) is in the relation if and only if implementation i is a conforming implementation of specification s.

We will use the relation **ioco** as implementation relation. This relation assumes that the specification is expressed as a labelled transition system in which inputs and outputs can be distinguished (not necessarily $\mathcal{IOTS}$), and that the implementation behaves as, i.e., can be modelled by, an input-output transition system (cf. *test hypothesis* [ISO96]): $\mathbf{ioco} \subseteq \mathcal{IOTS}(L_I, L_U) \times \mathcal{LTS}(L_I \cup L_U)$.

An implementation $i \in \mathcal{IOTS}(L_I, L_U)$ is **ioco**-correct with respect to the specification $s \in \mathcal{LTS}(L_I \cup L_U)$ if i can never produce an output which could not have been produced by s in the same situation, i.e., after the same trace. Moreover, i may only stay silent, i.e., produce no output at all, if s can do so. The absence of outputs is called *quiescence* and is denoted by a special label δ ($\delta \notin L \cup \{\tau\}$), cf. [Vaa91].

To formalize this notion of conformance **ioco** we first have to define quiescence as the absence of outputs. Then we have to extend traces of actions with the special action δ. Occurrence of δ in a state p, denoted by $p \xrightarrow{\delta}$, expresses that state p cannot produce any output. Since no 'normal' action in p is executed in that case, p cannot move to another state, so always $p \xrightarrow{\delta} p$. Traces in which both normal actions in L and the special action δ may occur are called *suspension traces*. To denote suspension traces the notations $\xrightarrow{\mu}$ and $\xRightarrow{\sigma}$ (definitions 2.1, 2.2 and 2.3) are extended to traces in $(L \cup \{\delta\})^*$. Note that this overlapping of notation does not introduce conflicts.

Definition 2.5
Let $p \in \mathcal{LTS}(L_I \cup L_U)$.

1. A state q of p is *quiescent*, denoted by $q \xrightarrow{\delta} q$, if $\forall \mu \in L_U \cup \{\tau\} : \ q \not\xrightarrow{\mu}$
2. The *suspension traces* of $p \in \mathcal{LTS}(L_I \cup L_U)$ are: $\mathit{Straces}(p) =_{\mathrm{def}} \{ \sigma \in (L \cup \{\delta\})^* \mid p \xRightarrow{\sigma} \}$, where $\xRightarrow{\sigma}$ is as in definition 2.2 extended with δ-transitions $q \xrightarrow{\delta} q$.

□

We can now define the possible outputs $\mathit{out}(\, p\, \mathbf{after}\, \sigma\,)$ of a process p after a suspension trace σ. The action δ may occur in $\mathit{out}(\, p\, \mathbf{after}\, \sigma\,)$ as a special action indicating that after σ it is possible to observe no outputs at all, i.e., quiescence. Using $\mathit{out}(\, p\, \mathbf{after}\, \sigma\,)$ the definition of **ioco** is now straightforward by requiring that after any suspension trace of the specification any possible output of the implementation should be a possible output of the specification.

Definition 2.6
Let p be a (state of a) labelled transition system, and let P a set of states; let $i \in \mathcal{IOTS}(L_I, L_U)$ and $s \in \mathcal{LTS}(L_I \cup L_U)$, then

1. $\mathit{out}(p) \ =_{\mathrm{def}} \ \{ \ x \in L_U \cup \{\delta\} \mid p \xrightarrow{x} \ \}$
2. $\mathit{out}(P) \ =_{\mathrm{def}} \ \bigcup \{ \ \mathit{out}(p) \mid p \in P \ \}$
3. $i \ \mathbf{ioco} \ s \quad =_{\mathrm{def}} \quad \forall \sigma \in \mathit{Straces}(s) : \quad \mathit{out}(\, i\, \mathbf{after}\, \sigma\,) \ \subseteq \ \mathit{out}(\, s\, \mathbf{after}\, \sigma\,)$

□

For more details about the relation **ioco**, for a rationale for its use, and for more generic definitions we refer to [Tre96].

Testing In order to generate and execute tests, we first have to define what a test case is, how test cases are executed, what a test run is, how a verdict is assigned and when an implementation passes a test case. We adopt, and adapt a little bit, the definitions of [Tre96].

Definition 2.7

1. Let $a \in L_I$ and $L_U = \{x_1, x_2, \ldots, x_n\}$. A *test case* t defined as

$$t \quad ::= \quad \mathbf{pass} \;\mid\; \mathbf{fail} \;\mid\; a; t \;\mid\; x_1; t \,\Box\, x_2; t \,\Box\, \ldots \,\Box\, x_n; t \,\Box\, \theta; t$$

   The special label $\theta \notin L \cup \{\tau, \delta\}$ will be used in a test case to detect quiescent states of an implementation, so it can be thought of as the communicating counterpart of a δ-action.

2. The finite sequence of pairs $(t_0, i_0) \cdot (t_1, i_1) \cdot (t_2, i_2) \cdot \ldots \cdot (t_m, i_m)$, representing the parallel computation of a test case t and an implementation i, is a *test run* of t and i iff

   - $t = t_0$ and $i = i_0$, and
   - $t_m = \mathbf{pass}$ or $t_m = \mathbf{fail}$, and
   - for all j with $1 \leq j \leq m$, either:
     - (* internal step *)
       $t_{j-1} = t_j$ and $i_{j-1} \xrightarrow{\tau} i_j$, or
     - (* offer input *)
       $t_{j-1} = a; t_j$ and $i_{j-1} \xrightarrow{a} i_j$, or
     - (* accept output *)
       $t_{j-1} = x_1; t_j^1 \,\Box\, \ldots \,\Box\, x_k; t_j^k \,\Box\, \ldots \,\Box\, x_n; t_j^n \,\Box\, \theta; t_j^\theta$ and $i_{j-1} \xrightarrow{x_k} i_j$ and $t_j = t_j^k$, or
     - (* accept quiescence *)
       $t_{j-1} = x_1; t_j^1 \,\Box\, \ldots \,\Box\, x_n; t_j^n \,\Box\, \theta; t_j^\theta$ and $i_{j-1} = i_j$ and
       $\forall \mu \in L_U \cup \{\tau\} : i_{j-1} \stackrel{\mu}{\not\longrightarrow}$ and $t_j = t_j^\theta$

3. An implementation i **passes** a test case t if all possible test runs of t and i end with $t_m = \mathbf{pass}$, otherwise i **fails** t. □

Test derivation Now all ingredients are there to present an algorithm to generate test cases from a labelled transition system specification for the implementation relation **ioco**.

Algorithm 2.8
Let s be a specification with initial state s_0. Let S be a non-empty set of states, with initially $S = s_0$ **after** ϵ. Then a test case t is obtained from S by a finite number of recursive applications of one of the following three nondeterministic choices:

1. ($*$ terminate the test case $*$)
   $t \;:=\; \mathbf{pass}$

2. ($*$ give a next input to the implementation $*$)
   $t \;:=\; a \;;\; t'$
   where $a \in L_I$, $S' = S$ **after** $a \neq \emptyset$, and t' is obtained by recursively applying the algorithm for S'.

3. ($*$ check the next output of the implementation $*$)
   $t \;:=\; x_1; t_1 \,\Box\, x_2; t_2 \,\Box\, \ldots \,\Box\, x_n; t_n \,\Box\, \theta; t_\theta$
   where, with $1 \leq j \leq n$:
   if $x_j \notin out(S)$ then $t_j = \mathbf{fail}$
   if $\delta \notin out(S)$ then $t_\theta = \mathbf{fail}$
   if $x_j \in out(S)$ then t_j is obtained by recursively applying the algorithm for S **after** x_j
   if $\delta \in out(S)$ then t_θ is obtained by recursively applying the algorithm for $\{s \in S \mid s \xrightarrow{\delta}\}$.

$\square$

This algorithm was proved in [Tre96] to produce only sound test cases, i.e., test cases which never produce **fail** while testing an **ioco**-conforming implementation. Moreover, it was shown that any non-conforming implementation can always be detected by a test case generated with this algorithm. Algorithm 2.8 will be the basis for test derivation from PROMELA specifications in the next sections.

3 On-the-Fly Testing

Derivation of test cases as explained in the previous section may involve the consideration of a large number of transitions and states. The encountered complexity is mainly due to the fact that in each state of the specification we have to consider all possible responses of the implementation. After this step, again all possible responses for all possible responses of the previous step have to be considered. Due to nondeterminism, parallelism, data instantiation and recursive processes there may be many different responses, and all these possible responses have to be recorded and represented in the generated test case. This is illustrated in example 3.2, but before giving this example, first an algorithm for test execution is presented. This algorithm, in fact, operationalizes the abstract concept of test run from definition 2.7.2 by giving the actions to be performed to obtain a test run.

Algorithm 3.1
Let t be a test case, $a \in L_I$, $x_j \in L_U$ and let $i \in \mathcal{IOTS}(L_I, L_U)$ be an implementation under test. Then test execution proceeds by the following rules:

WHILE $t \notin \{\textbf{pass}, \textbf{fail}\}$
{ apply one of the following choices:

- (∗ offer an input ∗)
  If $t = a\,; t'$ and $i \overset{a}{\Longrightarrow} i'$ then $t := t'; i := i'$
- (∗ accept quiescence ∗)
  If $t = x_1; t_1 \,\square\, \ldots \,\square\, x_n; t_n \,\square\, \theta; t_\theta$ and $\forall x \in L_U \cup \{\tau\} : i \overset{x}{\not\longrightarrow}$ then $t := t_\theta$
- (∗ accept output ∗)
  If $t = x_1; t_1 \,\square\, \ldots \,\square\, x_j; t_j \,\square\, \ldots \,\square\, x_n; t_n \,\square\, \theta; t_\theta$ and $i \overset{x_j}{\Longrightarrow} i'$ then $t := t_j;\ i := i'$

}

$\square$

Example 3.2
Consider a simple coffee-machine modelled by $s \in \mathcal{LTS}(L_I \cup L_U)$, where $L_I = \{coin\}$ and $L_U = \{coffee, tea\}$. The specification s of the coffee machine is given in Figure 1, together with a test case which has been derived from the specification using algorithm 2.8. The sequence of observable actions which can be observed during a possible test run of the test case with an implementation, which is able to execute the (erroneous) trace $coin \cdot tea \cdot tea$, is also given in Figure 1. The solid lines of the test run denote the actions that actually occurred during test execution. The dotted lines denote the actions that might have occurred during test execution, i.e., possible responses, but which did not actually occur. Since the end state is **fail**, we conclude that the tested implementation is not conforming.

After offering a *coin* to the coffee-machine, we have to consider all possible actions that could follow, i.e. θ, *coffee* and *tea*, which are partially marked by a dotted line in the test run. Since the system actually responds with a *tea* action after inserting the *coin*, we do not have to pay any attention to the subsequent behaviour of $coin{\cdot}\theta$ and $coin{\cdot}coffee$ in the test case. This behaviour

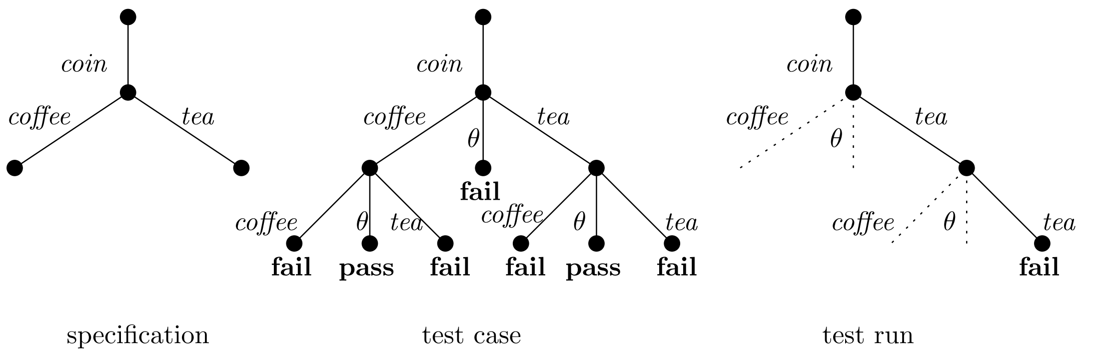

Figure 1: Execution of a test case

has been considered during the generation of the test case. We only have to resume consideration of the possible actions after *coin·tea* and continue test execution at that branch. When the next output *tea* is processed by the tester, the algorithm will terminate.

□

Example 3.2 shows the partial usage of the test case in the test execution Algorithm 3.1. The implementation under test produces in each state just one actual response, and all other responses which were considered during test derivation are not used, and were considered for nothing.

The aim of on-the-fly testing is to reduce the number of states and transitions to be considered by using the actual responses of the implementation under test. Only the part of the test case used during test execution is derived during on-the-fly testing. Of course, this implies that the actual responses of the implementation must be known during test derivation. Since the test run is not known beforehand, the derivation of the test case cannot be completed beforehand either. It should be done dynamically, during the execution of the test case.

From a certain state of the specification we need to derive the possible input actions, the expected output actions, and the possibility of quiescence. These are called *test primitives*. Let S be the set of states in which the specification may be after a particular partial test run, then these test primitives are in the set $(init(S) \cap L_I) \cup out(s)$.

The derivation of these test primitives from the specification while at the same time executing these actions is called on-the-fly testing. Intuitively, on-the-fly test execution can be characterized as a kind of feedback system in which information obtained during execution is fed back to test case derivation. Test cases are not explicitly generated and stored during on-the-fly testing.

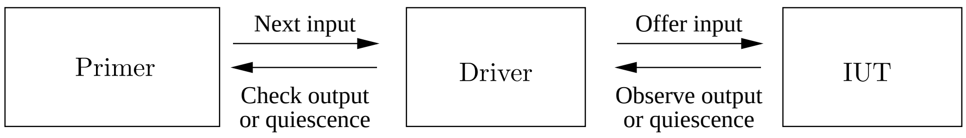

Figure 2: On-the-fly tester architecture

Figure 2 depicts schematically a possible architecture for an on-the-fly tester. The Primer analyses the specification and generates all the possible test primitives. It is an entity representing all possible current states of the specification taking into account the test actions executed during the current test run, including the responses from the implementation under test (IUT). Each time the Driver asks for the next test primitive from the Primer, it will immediately execute this primitive.

The Driver observes the responses from the IUT and feeds them back to the Primer.

Following the on-the-fly approach, the test case generation and test case execution algorithms, i.e, algorithms 2.8 and 3.1, have to be merged. This is done in Algorithm 3.3. By looking one observable transition ahead, Algorithm 3.3 is able to construct a sound test case during testing. This is the task to be done by the Driver. Based on the output generated by the IUT the Driver chooses one of the rules to be applied during execution.

Algorithm 3.3
Let initially TERMINATE = FAILURE = false; let s be the specification and let $S = s\ \mathbf{after}\ \epsilon$; let $i \in \mathcal{IOTS}(L_I, L_U)$ be the implementation under test. Then i is checked by application of the following rules. An implementation is **ioco**-conforming to the specification s, when the algorithm terminates with FAILURE = true. If the algorithm terminates with FAILURE = false, then we have one test run which does not produce fail, i.e., our confidence in the correct functioning of the implementation increases, although, formally, no judgement about conformance can be given.

WHILE not (TERMINATE or FAILURE)
{ apply one of the following choices:

1. (∗ offer an input ∗)
   Select an $a \in init(S) \cap L_I$, then $S := S\ \mathbf{after}\ a\,;\ i := i'$
   where $i \stackrel{a}{\Longrightarrow} i'$
2. (∗ accept quiescence ∗)
   If $\forall x \in L_U \cup \{\tau\} : i \stackrel{x}{\not\longrightarrow}$ and $\delta \in out(S)$ then $S := \{\ s \in S \mid s \stackrel{\delta}{\longrightarrow}\ \}$
3. (∗ fail on quiescence ∗)
   If $\forall x \in L_U \cup \{\tau\} : i \stackrel{x}{\not\longrightarrow}$ and $\delta \notin out(S)$ then FAILURE = true
4. (∗ accept output ∗)
   If $\exists x \in L_U : i \stackrel{x}{\Longrightarrow} i'$ and $x \in out(S)$ then $S := S\ \mathbf{after}\ x\,;\ i := i'$
5. (∗ fail on output ∗)
   If $\exists x \in L_U : i \stackrel{x}{\Longrightarrow} i'$ and $x \notin out(S)$, then FAILURE = true
6. (∗ terminate the loop ∗)
   TERMINATE = true

}

□

Example 3.4
Consider Figure 3 with the specification s of the coffee machine. An erroneous trace of implementation i, $coin{\cdot}tea{\cdot}tea$ is tested on-the-fly using algorithm 3.3. The choices within Algorithm 3.3 are successively ($S = \{s_0\}$ initially):

- choice 1: $a = coin \in init(S) \cap L_I$ and $S = S\ \mathbf{after}\ a = \{s_2\}$; $i := i_2$
- choice 4: since $tea \in L_U : i \stackrel{tea}{\Longrightarrow} i'$ and $tea \in out(S)$ then $S := S\ \mathbf{after}\ tea = \{s_4\}$; $i := i_4$
- choice 5: since $tea \in L_U : i \stackrel{tea}{\Longrightarrow} i_5$ and $tea \notin out(S)$ then FAILURE = true

Since FAILURE is true at termination of the algorithm, the implementation is not conforming to specification s.

□

Using algorithm 3.3, on-the-fly testing can be performed based on any specification formalism which can be expressed in labelled transition systems. The only thing which is needed is to develop a

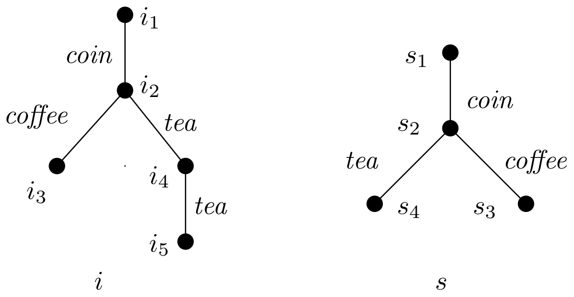

Figure 3: On the fly test execution example

Primer component which generates the test primitives captured by the sets $init(S) \cap L_I$ and $out(S)$, and means for state selection i.e., S **after** a for that specification formalism. Promela is such a formalism that can be expressed in labelled transition systems [Hol91]. Hence, the next section will discuss the derivation of the test primitives from Promela specifications using Spin.

4 Test Derivation for Promela

The developed test theory in the previous sections is based on the assumption that the underlying model of the specification is a labelled transition system (LTS). The underlying model of Promela is a composition of communicating finite state machines. The participating state machines communicate with each other and the environment by means of channels, i.e., finite queues. The state of the composite state machine is determined by the global context and the local context of each individual state machine. The context is determined by the variables and queue contents.

By making some restrictions and assumptions on the usage of the Promela model, it is possible to apply the theory developed in the previous sections. To do this, we assume that a Promela model can be considered as an LTS (Promela-LTS) from which the input and output operations on some channels are observable. The behaviour of the model is characterized by the sequences of input and output actions on these observable queues.

This means that, contrary to a Promela model used to validate a system, we have to enhance the specification for derivation of test primitives, in order to be able to distinguish between observable and non observable channels. A channel should be explicitly declared as an observable channel, for which we extended the Promela language with the keyword `observable`. Moreover, for testing there is no need to specify the environment within a specification, i.e., we do not use closed models, in contrast with the case of validation. For technical reasons due to the implementation of the Primer based on Spin, we insist that these observable channels are rendez-vous channels.

All the other actions occurring in the Promela model are mapped onto internal τ actions, including, e.g., assignments and actions on non-observable channels. Specifications with infinitely many outgoing transitions in one state are not allowed.

In order to obtain the test primitives at a particular stage of the testing process, we define the *super state* $S = s \,\textbf{after}\, \sigma \neq \emptyset$, where $\sigma \in L \cup \{\delta\}$. Intuitively, a super state S contains all the states in which the specified system can be after the partial test run σ. The test primitives at that super state are then $out(S) \cup (init(S) \cap L_I)$. We denote $S \xrightarrow{a} S'$ as the next super state after an action $a \in L \cup \{\delta\}$, i.e., $S' = S \,\textbf{after}\, a$. Obviously, $s_0 \,\textbf{after}\, \sigma{\cdot}a = \bigcup\{\, s' \,\textbf{after}\, a \,|\, s' \in s_0 \,\textbf{after}\, \sigma \,\}$, where s_0 is the initial state of specification s. These characterizations of the test primitives and super states are the basis of the operations required by Algorithm 3.3, e.g., performing a δ-, input-or output-transition from a super state to the next super state and obtaining the possibles actions at a super state.

Since we aim at developing an automatic test derivation tool based on SPIN we have some more requirements for an algorithm supporting these operations. Due to technical reasons of SPIN's generated machine representation, the transitions from a certain state are ordered. For instance when we consider a state (element of a super state) with several outgoing transitions $a_1, a_2 \dots a_n$, we inspect the actions in sequence from a_1 till a_n. The inspection of a transition using SPIN's representation to obtain the action associated with that transition involves actually making that transition, i.e., going to another state. A backward transition is then necessary to bring us back to the original state. By using this property, a depth-first search for test primitives and a new super state is in favour in order to reduce the computations by the algorithms. The presented algorithms are designed such that the initial super state $S = \{s_0\}$ is sufficient to be the root state.

An algorithm to determine the test primitives from a super state is presented in Algorithm 4.1. Example 4.2 shows the application of the algorithm to the specification s depicted in Figure 4.

Algorithm 4.1
Let S be a set of states of a PROMELA-LTS, and L_I and L_U are the input, respectively the output actions. Let the global set A initially be empty, being the result set with obtained test primitives. The set of test primitives in the super state S is obtained by application of the following rule:

For each $s \in S$ apply the following recursive procedure P(s, true)

Procedure P(s, δ)
Select each transition $s \xrightarrow{a} s'$ from the ordered list of transitions from state s and apply the following rules:

- if $a \in L_I$ then $A := A \cup \{a\}$
- if $a \in L_U$ then $\delta :=$ false and $A := A \cup \{a\}$
- if $a = \tau$ then $\delta :=$ false and apply recursively procedure P(s',true)

if $\delta =$ true then $A := A \cup \{\delta\}$

□

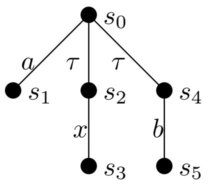

Figure 4: Specification s

Example 4.2
Consider Figure 4 where specification s with $L_I = \{a, b\}$ and $L_U = \{x\}$ is depicted. To obtain the test primitives of the initial state $S = \{s_0\}$ we apply the rules of Algorithm 4.1 as follows:

$S = \{s_0\}$ and $A = \emptyset$

1. call P(s_0,true)
2. select $s_0 \xrightarrow{a} s_1$; $A := A \cup \{a\} = \{a\}$
3. select $s_0 \xrightarrow{\tau} s_2$; $\delta =$ false; call P(s_2,true)
   (a) select $s_2 \xrightarrow{x} s_3$; $\delta =$ false; $A := \{a\} \cup \{x\} = \{a, x\}$

(b) $\delta =$ false do nothing

4. select $s_0 \xrightarrow{\tau} s_4$; $\delta =$ false; call P(s_4,true)
   (a) select $s_4 \xrightarrow{b} s_5$; $A = \{a, x\} \cup \{b\} = \{a, x, b\}$
   (b) $\delta =$ true so $A = \{a, x, b\} \cup \{\delta\} = \{a, x, b, \delta\}$
5. $\delta =$ false do nothing

Since the algorithm terminates the set of test primitives $A = \{a, x, b, \delta\}$.

□

An algorithm to perform a δ-transition on super states in the PROMELA-LTS, $S \xrightarrow{\delta} S'$, is presented in Algorithm 4.3.

Algorithm 4.3
Let S be a set of states of a PROMELA-LTS. L_I and L_U are the input, respectively the output actions. Let the global set Q initially be empty, being the resulting super state after a δ-transition. This super state Q is obtained by applying the following rule:

For each $s \in S$ apply the following recursive procedure P(s, true)

Procedure P(s, δ)
Select each transition $s \xrightarrow{a} s'$ from the ordered list of transitions from state s and apply the following rules:

- if $a \in L_I$ then nothing
- if $a \in L_U$ then $\delta :=$ false
- if $a = \tau$ then $\delta :=$ false and apply recursively procedure P(s',true)

if $\delta =$ true then $Q := Q \cup \{s\}$

□

An algorithm to make an input or output test primitive transition on super states in the PROMELA-LTS, $S \xrightarrow{a} S'$, with $a \in L$, is presented in Algorithm 4.4.

Algorithm 4.4
Let S be a super state of a PROMELA-LTS and $z \in L_I \cup L_U$ an input or an output primitive. Then the successor global super state vector Q (initially empty), $S \xrightarrow{z} Q$, is obtained by application of the following rule:

For each $s \in S$ apply the following recursive procedure P(s)

Procedure P(s)
Select each transition $s \xrightarrow{a} s'$ from the ordered list of transition(s) from state s and apply the following rules:

- if $a = z$ then $Q := Q \cup \{s'\}$
- if $a = \tau$ then apply recursively procedure P(s')

□

As a result of this study we have implemented these algorithms in a prototype called TROJKA. This prototype represents the primer module of the on-the-fly test architecture (Figure 2). This module is automatically generated by a modified version of SPIN, analogous to the generation of the PAN verification analyser from a PROMELA specification. Details about the implementation of this prototype can be found in [Vri96].

Although due to the on-the-fly testing principle we reduced the state space considerably, the state space still explodes during the search for test primitives. It helps, in order to reduce the number of calculations by Trojka, to use as much as possible the `d-step` or `atomic` construct of Promela to combine internal steps (τ). But more is needed. This leads to the application of the state matching principle of Spin. An already investigated state is not assessed again. In order to assure the soundness of the tester, we only apply state hashing with full comparison of states. An implementation may be judged incorrectly when the *out* set is incomplete.

It should be noticed that we should insist that the Promela-LTS is strongly converging (Section 2), in order to claim termination of the presented algorithms. But for practical reasons experienced during prototyping, we have also to handle (long) sequences of internal transitions, by cutting off the search at a certain depth. This solution is not very elegant since the tester looses the soundness property when cutting is applied. A more subtle solution like loop detection should be added to Trojka, cf. TGV [FJJV96].

We tested Trojka by interfacing it with an interactive application, which, in fact, simulates the Driver component. The results are very promising and it seems, despite the large number of states to be investigated, we can handle real specifications.

In the *Côte de Resyste* project [STW96, CdR98], we have implemented the whole on-the-fly tester for the specification language Lotos (i.e. Primer and Driver), which is able to test real systems. We have replaced the LOTOS-Primer by Trojka enabling us testing using Promela specifications. In this experiment we have performed some testing of an elevator based on a Promela specification. The results are similar to testing the elevator based on a LOTOS-specification.

5 Concluding Remarks

In this paper we discussed the on-the-fly conformance testing principle that is able to simultaneously generate test primitives and test an implementation. The implementation is tested on the correctness notion defined by **ioco**. The on-the-fly approach reduces the number of computations (consideration of states and transitions) during test derivation. Algorithms to derive test primitives from Promela models have been presented. The algorithms have been implemented in the Spin based tool Trojka.

At the moment the Trojka tool cannot produce test primitives which result in a granted sound test, when the number of internal steps is very large. A mechanism for τ-loop detection should be added to its functionality to remove this shortcoming. From experiments with large specifications (many transitions and states), we experienced that, despite the state space reduction, the state space is still very large. Additional mechanisms to reduce the state space should be investigated in future work, e.g., compositional test primitive derivation.

Within the *Côte de Resyste* project the whole test architecture is realized based on Lotos with well defined interfaces between the components. The Primer component used here is decomposed into two components; a (simpler) Primer (the test primitive deriver) and an Explorer. The latter is responsible for moving trough the transition system and inspecting transitions. The interface is based on the Open/Caesar interface [Gar98]. It would desirable to adapt Spin such that it supports the Open/Caesar interface, making it possible to integrate it in the whole on-the-fly tester. Another benefit could be that more tool support for Promela specifications becomes available.

Further study of input/output labelled transition systems has led to the definition of the implementation relation **mioco**. This implementation relation distinguishes between multiple channels and uses refusal of inputs [HT97, Hee98]. For future work we suggest adaptation of the algorithms in order to derive **mioco** test primitives, i.e., building a **mioco** on-the-fly conformance tester.

In practice, it is hard to do experiments with real systems since the pair specification - implementation is not available. In order to do coverage studies it is interesting to use Trojka with some additional control as a simulator of an implementation under test. In combination with the *Côte de Resyste* tester we could test a LOTOS specification against a PROMELA description used as implementation, and vice versa.

A final remark concerns the difference between a model used for validation and a specification used as the the basis for testing. Although in our approach both are expressed in PROMELA, it should be noted that a specification used to test an implementation should be complete, i.e., all the functional behaviour should be specified. A model usually is incomplete in that abstractions have been made. Hence, a validation model in PROMELA can usually not be used directly as a specification for testing. Further research for testing based on partial specifications, e.g., models, is suggested. This is also interesting with respect to test purposes (cf. [ISO91]).

Acknowledgement

The authors would like to acknowledge Axel Belinfante for his effort to interface TROJKA with the *Côte de Resyste* tester and his constructive comment.

References

[CdR98] Côte de Resyste consortium. Côte de resyste webpages, 1998. URL: `http://fmt.cs.utwente.nl/CdR`.

[FJJV96] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification CAV'96*. Lecture Notes in Computer Science 1102, Springer-Verlag, 1996.

[Gar98] H. Garavel. OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. In B. Steffen, editor, *Fourth Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, pages 68–84. Lecture Notes in Computer Science 1384, Springer-Verlag, 1998.

[Hee98] L. Heerink. *Ins and Outs in Refusal Testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1998.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall Inc., 1991.

[HT97] L. Heerink and J. Tretmans. Refusal testing for classes of transition systems with inputs and outputs. In T. Mizuno, N. Shiratori, T. Higashino, and A. Togashi, editors, *Formal Desciption Techniques and Protocol Specification, Testing and Verification FORTE X /PSTV XVII '97*, pages 23–38. Chapman & Hall, 1997.

[ISO91] ISO. *Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework*. International Standard IS-9646. ISO, Geneve, 1991. Also: CCITT X.290–X.294.

[ISO96] ISO/IEC JTC1/SC21 WG7, ITU-T SG 10/Q.8. *Information Retrieval, Transfer and Management for OSI; Framework: Formal Methods in Conformance Testing*. Committee Draft CD 13245-1, ITU-T proposed recommendation Z.500. ISO – ITU-T, Geneve, 1996.

[LT89] N.A. Lynch and M.R. Tuttle. An introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989. Also: Technical Report MIT/LCS/TM-373 (TM-351 revised), Massachusetts Institute of Technology, Cambridge, U.S.A., 1988.

[STW96] Dutch Technology Foundation STW. *Côte de Resyste* – COnformance TEsting of REactive SYSTEms. Project proposal STW TIF.4111, University of Twente, Eindhoven University of Technology, Philips Research Laboratories, KPN Research, Utrecht, The Netherlands, 1996.

[Tre96] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996. Also: Technical Report No. 96-26, Centre for Telematics and Information Technology, University of Twente, The Netherlands.

[Vaa91] F. Vaandrager. On the relationship between process algebra and Input/Output Automata. In *Logic in Computer Science*, pages 387–398. Sixth Annual IEEE Symposium, IEEE Computer Society Press, 1991.

[Vri96] R.G. de Vries. Conformance testing with Promela. Master's thesis, University of Twente, Enschede, The Netherlands, 1996.