An Approach to Testing with Embedded Context Using Model Checker

Lihua Duan and Jessica Chen

School of Computer Science, University of Windsor
Windsor, Ont. Canada N9B 3P4
{duan1,xjchen}@uwindsor.ca

Abstract. Testing each component in isolation is not always feasible. We consider FSM-based deterministic testing on an Implementation Under Test (IUT) together with some other correctly implemented components as its context. The behavior of the context needs to be taken into account for generating test sequences. We employ model checking tools to retrieve necessary information from the context specification so that a test suite for the IUT integrated with its context can be generated. The use of model checking tools frees us from the necessity of constructing the global model of the IUT and its context, and thus helps avoid the state explosion problem. In the current work, we consider the situations when the context is an embedded system, i.e. it communicates and only communicates with the IUT. In this setting, we present a method to derive a *complete* test suite that can be used to check for trace pre-order between the FSM representing the integrated implementation of the IUT and its context and the synchronous product of the specification FSM of the IUT and that of its context.

Keywords: finite state machines, conformance testing, context-based testing, test sequences, distinguishing sequences.

1 Introduction

Given a final software product, we are interested in knowing whether it *conforms* to what we expect. *Conformance testing* has been extensively studied and has turned out to be an effective tool for us to gain enough confidence in the correctness of our product implementation with respect to the expectations. It has two main characteristics: (i) Different from a formal verification approach, here the *implementation* (called *implementation under test* (IUT)) is treated as a *black box* from which we can only infer its behavior by providing input to it and observing its output. (ii) Instead of simple input/output pairs, the expected behavior in different states is formally specified. This is because in most of the cases, the given IUT is *stateful* in the sense that it reacts differently (e.g. by giving different outputs) to the same input provided at different time of the execution.

There are various tools to describe the expected behavior in different states. Suitable for different levels of abstractions, they range from formal specification languages such as process algebras, to structural/operational modelling languages such as (input/output) *labelled transition systems (LTSs)* and *Finite State Machines (FSMs)*. Of course, when a specification is given in some high level formal specification languages, we may still refer to its formal model.

Given a formal model $\mathcal{S}$ in terms of e.g. LTS or FSM, describing the expected behavior of the IUT, we can imagine that the IUT behaves according to a certain abstract machine $\mathcal{M}$ in the same format. In this setting, conformance testing amounts to establishing the correspondence between $\mathcal{S}$ and $\mathcal{M}$. There are several relations proposed in the literature in this regard: the *trace equivalence* relation, the *ioco conformance* relation [1], the *quasi-equivalence relation* [2], etc. In this work, we consider *trace pre-order* $\preceq$. $\mathcal{S} \preceq \mathcal{M}$ holds if any (input/output) trace allowed by $\mathcal{S}$ are implemented, yet a trace not specified in $\mathcal{S}$ may or may not be implemented. In a special case when the specifications of the IUT and its context are both completely specified, our results may hold for *trace equivalence*.

There is usually an infinite number of traces in a given model of the desired behavior and the one representing the IUT, some of them with infinite lengths. The ultimate goal for test generation is to find a sufficient and efficient set of finite input sequences, i.e. a test suite, from a given model so that when these input sequences are given to the IUT, we can, by comparing the actual output sequences with the expected ones, draw a conclusion whether the trace pre-order holds between the model of the expected behavior and the one representing the IUT.

This goal can be better achieved with the *slow environment* assumption well-adopted in the literature, i.e., whenever an input reaches the system, the system will always prompt the output for it and reach a *stable state* (i.e. there is no more executable statement from that state) before the next input can reach the system. In other words, each input is explicitly associated with one or more outputs. In this setting, under the assumption that the given model and the one representing the IUT share the same sets of inputs and outputs, we may be able to *identify* a state in the model that represents the IUT by observing a sequence of outputs in response to a special sequence of inputs. In protocol testing, people used FSMs to represent the state changes with the paired input and output, and have explored the characteristics of such special input sequences as expressed in the notions of *characterization set* [3], *Unique Input/Output sequence* (UIO) [4], *distinguishing sequence* [5]. With a *characterization set*, a set of *UIO sequences*, or a *distinguishing sequence*, we can identify the states in the implementation FSM with those in the specification FSM, based on which we can further *verify* the correspondence of the transitions in the specification FSM with those in the implementation FSM. This helps us establish an equivalence or pre-order relation between the specification FSM and the implementation FSM.

One of the major drawbacks of FSM-based testing is that the specification FSM, possibly derived from a specification given in a higher level of abstraction, suffers from the state explosion problem: it may have too large a state space even if we only consider the *control data*. This is troublesome especially when we consider testing more general software systems than communications protocols. A promising solution to this problem is found in the compositional testing approach: we can apply FSM-based testing techniques for unit testing, while leaving the correctness of the integrated system to formal verification. Of course, this is a sound approach only if the considered equivalence or pre-order relation is *compositional*, e.g., if the specifications representing components $\mathcal{I}_1$ and $\mathcal{I}_2$ are X-equivalent to their respective behavioral specifications $\mathcal{S}_1$ and $\mathcal{S}_2$, then without performing *integration testing* or *system testing*, we know that the integration of $\mathcal{I}_1$ and $\mathcal{I}_2$ is a correct implementation of the parallel composition of $\mathcal{S}_1$ and $\mathcal{S}_2$ with respect to this X-equivalence. As we know, trace equivalence and trace pre-order are compositional when $\mathcal{I}_1$ and $\mathcal{I}_2$ are sequentially executed components represented by *deterministic* FSMs. The condition for *ioco* conformance relation to be compositional is given in [6].

Along the compositional testing approach to applying FSM-based testing techniques to unit testing, we may still encounter difficulties. One of them comes from the fact that testing each component in isolation is not always feasible. There are situations when we have to test a component together with some others.

As pointed out in [2], this can be the case when the IUT is an embedded part of a complex *system under test*. As another example, suppose we want to test a web-based composite service implementation $\mathcal{I}_1$. $\mathcal{I}_1$ makes use of another component service $\mathcal{I}_2$ which is known to be correct. When testing $\mathcal{I}_1$ in isolation, we have the difficulty in providing input and observing output all encapsulated according to certain protocol such as SOAP. Testing $\mathcal{I}_1$ in isolation also invokes the necessity of testing the interoperability between $\mathcal{I}_1$ and $\mathcal{I}_2$. A better solution is to test $\mathcal{I}_1$ and $\mathcal{I}_2$ together where $\mathcal{I}_2$ is considered as the *context* of $\mathcal{I}_1$.

Following the framework presented in [2] on *testing in context*, we consider the problem of FSM-based deterministic testing on $(\mathcal{I}, \mathcal{I}_c)$ which is an IUT implementation $\mathcal{I}$ together with a correct context implementation $\mathcal{I}_c$. In our current work, $\mathcal{I}_c$ is an embedded system, i.e. it does not communicate with any component other than $\mathcal{I}$. The communication port between $\mathcal{I}$ and $\mathcal{I}_c$ is not *controllable* but *observable*. This means that the tester can neither provide input to the IUT using this port nor stop an input from the context to the IUT. It can, however, observe all the input from and all the output to the context. The specification of $\mathcal{I}$ is given in terms of a *2-port FSM*, one port for communicating with its context and one for its input/output with the environment/tester. The specification of $\mathcal{I}_c$ can be given in terms of either a specification language or a structural modelling language. We present a method to generate a suitable test suite that can be used to test $(\mathcal{I}, \mathcal{I}_c)$. More precisely, we modify the well-known W-method [3] to construct test sequences in order to establish trace pre-order between the FSM representing $(\mathcal{I}, \mathcal{I}_c)$ and the product of the specification FSM of $\mathcal{I}$ and that of $\mathcal{I}_c$. The ultimate goal of our work is to avoid generating the operational model of the give specification of $\mathcal{I}_c$ (if a higher level specification is provided) and constructing the global model of $\mathcal{I}$ and $\mathcal{I}_c$. In order to do so, we employ model checking tools to retrieve necessary information from the context specification so that test sequences for $(\mathcal{I}, \mathcal{I}_c)$ can be generated. In this way, we avoid the notorious state explosion problem. Note that it is straightforward to extend our work to a more general case where the embedded context consists of a set of components, each having its own port to communicate with $\mathcal{I}$.

The rest of the paper is organized as follows. In Section 2 we give a brief notational background introduction to FSM and FSM-based testing that will be used later on. Our problem is explained in detail in Section 3, followed by a guideline of a possible solution. How to use model checking tools to generate a complete test suite is presented in Section 5. To better illustrate our method, we give a running example in Section 6. At the end, we position our work among other pieces of related work, and conclude ours with some final remarks.

2 Notational Background

In this section, we introduce the preliminary notations and terminologies on n-port finite state machines and test sequence construction. They will be used later in this paper.

2.1 n-Port Finite State Machines

As we mentioned in the Introduction, we assume that the specification $\mathcal{S}$ of the IUT is given in terms of a 2-port FSM. A deterministic *n-port Finite State Machine* (also called *finite state machine* for short) is defined by a tuple $(S, I, O, \delta, \lambda, s_0)$.

- S is a finite set of states where $s_0 \in S$ is its initial state.
- $I = \bigcup_{i=1}^{n} I_i$, where I_i is the input alphabet of port i $(i = 1, \ldots, n)$.
  Being abstract, these input symbols encapsulate the information of the communication channels. Thus, without loss of generality, we can assume that the input symbols at different ports are distinct, i.e. $I_i \cap I_j = \emptyset$ for $i \neq j$.
- $O = \Pi_{i=1}^{n} O_i$ where O_i is the output alphabet of port i $(i = 1, \ldots, n)$.
  Each $o \in O$ is a *vector of outputs* denoted by $o = \langle o_1, \ldots, o_n \rangle$ where $o_i \in O_i$ for $i = 1, \ldots, n$. We do not consider the order in which we observe output o_i and o_j at different ports. When there is no output at a port i, we use a special and distinct symbol $-$ to denote it.
- δ is the transition function that maps $S \times I$ to S, and λ is the output function that maps $S \times I$ to O.

The input and output symbols are abstract: The discussions on data types and complicate data structures in the input and the output are not considered.

Note that λ and δ are partial functions. We will use $\delta(s, i) = \mathit{null}$ to denote that there is no image of δ for the given state s of S and the given input i of I. In this case, we also have $\lambda(s, i) = \mathit{null}$. Furthermore, we extend the input of λ and δ from an input alphabet to a sequence of input alphabets with their meanings obtained straightforwardly from the original ones.

A *transition* t is defined by a tuple $(s_1, s_2, i/o)$ in which s_1 is the *starting state*, i is the input, $s_2 = \delta(s_1, i)$ is the *ending state*, and $o = \lambda(s_1, i)$ is the output. The input/output i/o is called the *label* of t. We use T to denote the set of all transitions in $\mathcal{S}$.

Let t_i be a transition for $1 \leq i \leq k$. A *path* $\rho = t_1\ t_2\ \ldots\ t_k$ is a finite sequence of transitions such that for $k \geq 2$, the ending state of t_i is the starting state of

t_{i+1} for all $1 \leq i \leq k-1$. A state $s \in S$ is *reachable* if there exists a path starting from s_0 and ending at s. We consider FSMs where all states are reachable.

An FSM is *completely specified* if functions λ and δ are *total*; otherwise, it is *partially specified*. We consider that the given specification FSMs are partially specified. Results in our approach when a given specification is completely-specified are discussed at the end. Note that if an FSM $\mathcal{S}$ is not completely specified, it is possible to make $\mathcal{S}$ completely specified by adding transition $(s, s, i/\langle -, \ldots, - \rangle)$ for each $s \in S, i \in I$ such that $(s, i) \notin \mathit{domain}(\delta)$. This, however, slightly changes the meaning of the FSM and is not always acceptable.

Two states s_i and s_j are *equivalent* if, for every input sequence σ, $\lambda(s_i, \sigma) = \lambda(s_j, \sigma)$. If $\lambda(s_i, \sigma) \neq \lambda(s_j, \sigma)$ then σ *distinguishes* between s_i and s_j. An FSM M is *minimal* if every state can be reached from the initial state of $\mathcal{M}$ and no two states of $\mathcal{M}$ are equivalent. Since only deterministic FSMs are considered, we can easily obtain a minimal FSM from any given FSM. In the following, we assume that all given FSMs are minimal.

2.2 Distinguishing Sequence and Test Sequence Construction

Let $\rho = (s_1, s_2, i_1/o_1)(s_2, s_3, i_2/o_2) \ldots (s_k, s_{k+1}, i_k/o_k)$ $(k \geq 1)$ be a path in an FSM. We will use $is(\rho)$ to denote the input sequence $i_1 \circ i_2 \circ \ldots i_k$ of ρ. Note that for clarity, we use $\circ$ as a separator in a sequence of input, a sequence of output, or a sequence of input/output pairs. A *test sequence* is an input sequence and is typically obtained from a path of a given specification FSM. A test suite is a finite set of finite test sequences. Usually, we assume that the IUT can always be reset to its initial state from any state and thus a test suite refers to a set of input sequences derived from paths that start from the initial state s_0. In this way, we can carry out the test with the test sequences in a test suite one by one. For each input sequence σ in a test suite, we will use $\mathit{path}(\sigma)$ to denote the path that σ is derived from. We will also use $\mathit{out}(\sigma)$ to denote the expected output sequence which is actually $\lambda(s_0, \sigma)$.

Given an FSM $\mathcal{S}$, we are interested in the so-called *complete test suites w.r.t. trace pre-order*. That is, by applying its input sequences to the IUT and comparing the output sequences with the expected ones, we can distinguish any implementation FSM $\mathcal{M}$ of the IUT if $\mathcal{S} \not\precsim M$.

Since $\mathcal{S}$ and $\mathcal{M}$ are deterministic and minimal, this can be achieved by establishing correspondence between the states in $\mathcal{S}$ and those in $\mathcal{M}$. Then, for each transition $t = (s_1, s_2, i/o)$ in $\mathcal{S}$, we construct a test sequence to verify that there exists a transition $t' = (r_1, r_2, i/o)$ in $\mathcal{M}$ which starts from a state corresponding to s_1, ends at a state corresponding to s_2, and gives the same output o upon the same input i.

The states in the implementation FSM can be identified via *distinguishing sequence*, *Unique Input/Output sequence (UIO)*, or *characterization set*. State identification using UIOs is possible but it turns out to be hard and less practical [7]. A characterization set is easier to find than a distinguishing sequence, yet a test suite generated using a characterization set [3] is usually much longer than that generated using a distinguishing sequence in terms of total length of the test sequences [5, 8–10]. Here we consider using distinguishing sequence.

A *distinguishing sequence* is an input sequence D with the following characteristics: the output sequences produced by $\mathcal{S}$ in response to D in different states of $\mathcal{S}$ are all different, i.e., for all $s_i, s_j \in S$, if $s_i \neq s_j$ then $\lambda(s_i, D) \neq \lambda(s_j, D)$.

There are various methods proposed in the literature for generating test sequences in order to check if an IUT conforms to a given specification FSM. See [11] for a survey on this topic. A basic idea of constructing a complete test suite $\mathcal{T}$ w.r.t. trace pre-order with a given distinguishing sequence D can be described as follows.

– For each state s_k in specification FSM $\mathcal{S}$, find a path ρ_k starting from s_0 and ending at s_k, and add a test sequence $is(\rho_k) \circ D$ to $\mathcal{T}$. If the IUT passes test sequence $is(\rho_k) \circ D$, i.e. its output sequence in response to this input sequence is correct, then we say that the state of the IUT after applying *$is(\rho_k)$ corresponds to s_k*. As we assume that the number of states in the implementation FSM $\mathcal{M}$ is no more than that in $\mathcal{S}$, set $\{is(\rho_k) \circ D \mid s_k \in S\}$ actually helps us to establish a one-to-one correspondence between the states in $\mathcal{S}$ and those in $\mathcal{M}$.
– For each transition $(s_1, s_2, i/o)$ in $\mathcal{S}$, add test sequence $is(\rho_1) \circ i \circ D$ to $\mathcal{T}$. Since $is(\rho_1) \circ D \in \mathcal{T}$, we know that the state of $\mathcal{M}$ after applying $is(\rho_1)$ corresponds to s_1. Thus, $is(\rho_1) \circ i \circ D$ helps us to check that there exists a transition $(r_1, r_2, i/o)$ in $\mathcal{M}$ where r_1 and r_2 correspond to s_1 and s_2 respectively: We can verify whether the state of $\mathcal{M}$ after applying $is(\rho_1) \circ i$ corresponds to s_2 by applying D on it. This is because from the first step, we have used the same distinguishing sequence D to identify all the states in $\mathcal{M}$. We say that transition $t = (s_1, s_2, i/o)$ is *verified* in a test suite $\mathcal{T}$ if there exists an input sequence σ such that $\sigma \circ D \in \mathcal{T}$, $\sigma \circ i \circ D \in \mathcal{T}$ and $path(\sigma)$ is a path in $\mathcal{S}$ from s_0 to s_1.

This is actually a variation of Chow's W-method [3] in the case when (i) the number of states in the implementation FSM is no more than that in the specification FSM; and (ii) a distinguishing sequence rather than a characterization set is available. Thus, it is straightforward that a test suite such constructed is complete with repect to trace pre-order.

In the following, we present an extension of this method for testing in context.

3 Problem Description

As noticed in [2], testing an IUT in isolation is quite different from testing it within a context. First of all, if an IUT is tested within a context and passed a test, we cannot draw any conclusion about the correctness of the IUT because a fault in the IUT and a fault in its context may mask each other resulting in an overall correct execution. Such a problem is out of the scope of our current work. In the following, we consider that the context is correctly implemented, with its behavior specified in $\mathcal{C}$.

Recall that in our setting, the FSM for an IUT has two ports: one for communicating with its context, called the *context port*; and the other for communicating with the rest part of its environment simulated by a tester, called *environment port*. For clarity, we will use

– I and O as the IUT's input and output at the environment port;
– X and Y as the IUT's input and output at the context port.

The behavior of the IUT is thus given as $\mathcal{S} = \langle S, s_0, I \cup X, O \cup Y, \lambda_s, \delta_s \rangle$.

We assume that the specification FSM $\mathcal{S}$ is free from *internal-port-cycles*. An *internal-port-cycle* in an FSM is a path $(s_1, s_2, i_1/o_1)$ $(s_2, s_3, i_2/o_2) \ldots (s_k, s_{k+1}, i_k/o_k)$ $(k \geq 2)$ such that $s_1 = s_{k+1}$, and $i_j \notin I$ for all $1 \leq j \leq k$. An *internal-port-cycle* represents a possibly infinite internal communications between the IUT and its context, which is normally considered as a design error. How to guarantee that the design specifications are free from such logical errors can be carried out by formally verifying the correctness of the design specifications.

An input sequence generated from $\mathcal{S}$ cannot be served as an input sequence to test the IUT in its context $\mathcal{I}_c$, as we cannot control the IUT's context port. To take the context into consideration, a possible approach is to develop a testing technique to check whether $\mathcal{M}$ conforms to $\mathcal{S}$ *within context* $\mathcal{C}$ w.r.t. trace pre-order, instead of checking whether $\mathcal{M}$ conforms to $\mathcal{S}$ w.r.t. trace pre-order. That is, we compare the model representing the actually behavior of $(\mathcal{I}, \mathcal{I}_c)$ with the one specifying its expected behavior. Just like we assume that the actual behavior of the IUT can be described by an FSM for testing the IUT in isolation, we assume that the actual behavior of $(\mathcal{I}, \mathcal{I}_c)$ can be described by an FSM.

The model representing the expected behavior of $(\mathcal{I}, \mathcal{I}_c)$ can be derived from the specification of the IUT and that of the context. Suppose that the context specification $\mathcal{C}$ is given as a 1-port FSM. Of course, if it is given in a specification language with higher level of abstraction, we consider its equivalent FSM model. Let

$$\mathcal{C} = \langle C, c_0, \bar{Y}, \bar{X}, \lambda_c, \delta_c \rangle$$

be the specification FSM of the context where $\bar{X} = \{\bar{x} \mid x \in X\}$ and $\bar{Y} = \{\bar{y} \mid y \in Y\}$ are the output and input symbols of $\mathcal{C}$ to communicate with $\mathcal{S}$: $\bar{x}$ and $\bar{y}$ are executed simultaneously with x and y respectively, representing the communications between the IUT and its context. Here we have ignored those actions internal to the context component.

Note that since we have the slow environment assumption, it makes no difference to use synchronous or asynchronous communication mode between the IUT and its context. For simplicity, we consider synchronous communication.

Given $\mathcal{S}$ and $\mathcal{C}$ as the above defined 2-port and 1-port FSMs, the expected behavior of $(\mathcal{I}, \mathcal{I}_c)$ can be described as a synchronous product FSM $\mathcal{S} \times \mathcal{C}$ defined on $\mathcal{S}$ and $\mathcal{C}$ as $\langle S', (s_0, c_0), I, ((O \times Y) \cup X)^*, \lambda, \delta \rangle$. It has only one port with the tester/environment for input. A global state consists of a local state of $\mathcal{S}$ and a local state of $\mathcal{C}$. $S' \subseteq S \times C$ is a set of global states reachable from (s_0, c_0) in the sense that for any $(s, c) \in S'$, there exists an input sequence $\sigma \in I^*$ such that $\delta((s_0, c_0), \sigma) = (s, c)$.

$((O \times Y) \cup X)^*$ is a set of outputs from the tester's viewpoint. As we mentioned in the Introduction, we assume that even though the input/output between the

IUT and its context is not controllable, they are observable. Thus, corresponding to each input from the environment, the tester will observe a sequence of outputs which is composed of those outputs $\langle o, y \rangle$ of the transitions in $\mathcal{S}$ ($\langle o, y \rangle \in O \times Y$) and those input x from its context ($x \in X$).

A transition in $\mathcal{S} \times \mathcal{C}$ is derived from a path in $\mathcal{S}$ and a path in $\mathcal{C}$. More precisely, we have transition $((s_1, c_1), (s_2, c_2), i/o)$ in $\mathcal{S} \times \mathcal{C}$, and thus $\lambda((s_1, c_1), i) = o$ and $\delta((s_1, c_1), i) = (s_2, c_2)$, only if we have

$$\lambda_s(s_1, i_1 \ldots i_k) = o_1 \ldots o_k,\ \delta_s(s_1, i_1 \ldots i_k) = s_2,$$
$$\lambda_c(c_1, i'_1 \ldots i'_h) = o'_1 \ldots o'_h,\ \delta_c(c_1, i'_1 \ldots i'_h) = c_2;$$

for $h, k \geq 1$ such that

$$k = h,\ i = i_1,\ o = o_1 \circ i_2 \circ o_2 \ldots \circ i_k \circ o_k,$$
$$i'_j = \overline{c(o_j)} \text{ for } 1 \leq j \leq k,\ i_{j+1} = \overline{o'_j} \text{ for } 1 \leq j \leq k-1,\ o'_k = -;$$

or

$$k = h+1,\ i = i_1,\ o = o_1 \circ i_2 \circ o_2 \ldots \circ i_k \circ o_k,$$
$$i'_j = \overline{c(o_j)} \text{ for } 1 \leq j \leq k-1,\ i_{j+1} = \overline{o'_j} \text{ for } 1 \leq j \leq k-1,$$
$$o_k = \langle *, - \rangle \text{ where * can be any output including -;}$$

Otherwise, $\lambda((s_1, c_1), i) = null$ and $\delta((s_1, c_1), i) = null$. Here $c(o)$ represents the output of o at the context port. Note that in the following, when there is no confusion, we will drop the subscripts of λ and δ.

Since there is no *internal-port-cycle* in $\mathcal{S}$, the above defined product FSM fully describes the expected behavior of the IUT with its context using the slow environment feature. Furthermore, as we assume that $\mathcal{S}$ and $\mathcal{C}$ are minimal and deterministic, the above defined synchronous product of them is also minimal and deterministic.

Once we have a product FSM specification for the expected behavior of $(\mathcal{I}, \mathcal{I}_c)$, it is straightforward to generate a suitable test suite from this product FSM in order to test whether trace pre-order holds between this specification and the implementation FSM of $(\mathcal{I}, \mathcal{I}_c)$.

This approach, however, requires that the FSM specification of $\mathcal{I}_c$ be available, and the global model of $(\mathcal{I}, \mathcal{I}_c)$ be calculated, which brings out the state explosion problem. In the present work, we consider using model checker as an auxiliary tool to retrieve necessary information from a context specification in order to generate test sequences. We do not require that the product of $\mathcal{S}$ and $\mathcal{C}$ be actually constructed. In particular, if the specification of the expected behavior of $\mathcal{I}_c$ is given in a specification language of a higher level of abstraction, we do not need to construct its operational model neither.

4 Test Generation with Context

To check whether a trace pre-order relation holds between $\mathcal{S} \times \mathcal{C}$ and the implementation FSM of $(\mathcal{I}, \mathcal{I}_c)$, according to what we introduced in Section 2, we need to generate a complete test suite to *identify* all the states in $\mathcal{S} \times \mathcal{C}$ using a distinguishing sequence, and *verify* all the transitions in $\mathcal{S} \times \mathcal{C}$ using the same distinguishing sequence. Since the context implementation is known to be correct, we actually only need to generate test sequences to verify *some* of the transitions in $\mathcal{S} \times \mathcal{C}$. Consequently, we can look for a distinguishing sequence that is capable of distinguishing only a subset of states in $\mathcal{S} \times \mathcal{C}$. In this section, we characterize such a subset of transitions and a subset of states.

Definition 1 ($\mathcal{R}$ covers T). *Let T be the set of transitions in $\mathcal{S} \times \mathcal{C}$, and $\mathcal{R} \subseteq T$. $\mathcal{R}$ covers T if for any transition $((s_1, c_1), (s_2, c_2), i/o) \in T$, there exists a transition $t = ((s_1, c_1'), (s_2, c_2'), i/o)$ in $\mathcal{R}$ where $(s_1, c_1), (s_2, c_2), (s_1, c_1')$, and (s_2, c_2') are states in $\mathcal{S} \times \mathcal{C}$, i is an input of $\mathcal{S} \times \mathcal{C}$ and o is an output of $\mathcal{S} \times \mathcal{C}$.*

The transitions in $\mathcal{S} \times \mathcal{C}$ can be partitioned into different groups according to the local states of $\mathcal{S}$ in their starting states, the local states of $\mathcal{S}$ in their ending state, and their input/output pairs. The above definition actually requires that the subset of transitions $\mathcal{R}$ contain at least one representative transition from each of the partitions. The intuition behind is this: Since $\mathcal{S}$ and $\mathcal{C}$ are deterministic, given two states s_1 and s_2 in $\mathcal{S}$, an input i and an output o in $\mathcal{S} \times \mathcal{C}$, there exists exactly one path ρ in $\mathcal{S}$ from s_1 to s_2 with input/output sequence $i_1/o_1 \circ i_2/o_2 \circ \ldots \circ i_k/o_k$ such that $i = i_1$ and $o = o_1 \circ i_2 \circ o_2 \circ \ldots \circ i_k \circ o_k$. According to the definition of synchronous product, for any states c_1, c_2 in $\mathcal{C}$, if transition $t = ((s_1, c_1), (s_2, c_2), i/o) \in T$, then t is constructed from this path. Consider all such transitions in one partition $G(s_1, s_2, i, o)$. To check that each transition in $G(s_1, s_2, i, o)$ is correctly implemented, we only need to make sure that path ρ is correctly implemented in the sense that there exists a path ρ' in $\mathcal{M}$ which starts from a state identified as s_1, ends at a state verified as s_2, and correctly gives output o in responds to input i. Since the context is correct, this implies that all transitions in partition $G(s_1, s_2, i, o)$ are correctly implemented. While any transition in $G(s_1, s_2, i, o)$ can be used to generate a test sequence for the above purpose, we require that the subset $\mathcal{R}$ of transitions contains one transition from each partition $G(s_1, s_2, i, o)$.

As we consider only transitions in such a subset of transitions $\mathcal{R}$ that covers the total set of transitions in $\mathcal{S} \times \mathcal{C}$, we only need a distinguishing sequence to identify all the states appeared as the starting or ending states in the transitions in $\mathcal{R}$, denoted by $states(\mathcal{R})$. In the following, we show that we can further weaken this requirement: it is sufficient to have a distinguishing sequence that can identify, among the states in $states(\mathcal{R})$, all those with different local states of $\mathcal{S}$.

Definition 2 (distinguishing sequence on $\mathcal{S}$ over $\mathcal{W}$). *Let $\mathcal{W}$ be a subset of reachable states in $\mathcal{S} \times \mathcal{C}$. An input sequence $D = i_1 \circ \tilde{x}_1 \circ i_2 \circ \tilde{x}_2 \ldots \circ i_k \circ \tilde{x}_k$ for $i_j \in I$, $\tilde{x}_j \in X^*$ $(1 \leq j \leq k)$ is a distinguishing sequence on $\mathcal{S}$ over $\mathcal{W}$ if*

- *For any state s, $s' \in S$, $s \neq s'$ implies $\lambda(s, D) \neq \lambda(s', D)$.*
- *For any $(s_1, c_1) \in \mathcal{W}$ and for any h $(1 \leq h \leq k)$, the input sequence of X^* obtained from $\lambda((s_h, c_h), i_h)$ by removing all output of Y is $\tilde{x}_h$. Here for $2 \leq h \leq k$, $(s_h, c_h) = \delta((s_1, c_1), i_1 \circ i_2 \ldots \circ i_{h-1})$.*

The above definition can be viewed as an extension of the normal definition of distinguishing sequence of an FSM: A distinguishing sequence of $\mathcal{S}$ over $\emptyset$ is actually the original definition of distinguishing sequence on $\mathcal{S}$ without considering any context.

Note that we do not require an input sequence to distinguish all the states in $\mathcal{S} \times \mathcal{C}$, but a subset of states of interest expressed in $\mathcal{W}$. This brings out two benefits: i) an increased possibility of the existence of a distinguishing sequence; ii) when there exist distinguishing sequences, a possibly shorter one which contributes to the reduction of the cost for carrying out the test.

Now we show that in order to generate from $\mathcal{S} \times \mathcal{C}$ a complete test suite w.r.t. trace pre-order, it is sufficient to consider a subset $\mathcal{R}$ of transitions as long as $\mathcal{R}$ covers its set T of transitions, and a distinguishing sequence on $\mathcal{S}$ over *states*$(\mathcal{R})$.

Note that while previous work on this topic for testing in isolation requires *reliable reset*, i.e. the IUT can be reset to its initial state at any time, here we assume that the IUT can be reset to its initial state at any time and its context will be reset at the same time.

Similar to previous work, we assume a bound on the number of states in the implementation FSM of the IUT. When we test an IUT with a context, since the input to the IUT from the context is not *controllable*, the description of the IUT can be considered as a 1-port FSM from the tester's viewpoint. As a consequence, some of the states in a given 2-port FSM are not *stable* (so-called *transient states* in [2]) in the sense that after an input from the tester/environment, the IUT will never stay in any of those states waiting for the next input from the tester/environment. For testing in context, we consider only stable states: When we say that the number of states in the implementation FSM of the IUT is no more than the number of states in the specification FSM of the IUT, we refer to those states that appear to be the starting states of some transitions with input at the environment port.

With the above assumptions, we present the following result:

Proposition 1. *Let T be the set of transitions in $\mathcal{S} \times \mathcal{C}$ and $\mathcal{R} \subseteq T$. Let $\mathcal{T}$ be a test suite derived from $\mathcal{S} \times \mathcal{C}$. If*

- *$\mathcal{R}$ covers T,*
- *there exists an input sequence D such that D is a distinguishing sequence on $\mathcal{S}$ over states$(\mathcal{R})$, and $\forall t = ((s_1, c_1), (s_2, c_2), i/o) \in \mathcal{R}$, there exists an input sequence σ such that $\sigma \circ D \in \mathcal{T}$, $\sigma \circ i \circ D \in \mathcal{T}$, and* $\mathrm{path}(\sigma)$ *is a path in $\mathcal{S} \times \mathcal{C}$ from (s_0, c_0) to (s_1, c_1),*

then $\mathcal{T}$ of $\mathcal{S} \times \mathcal{C}$ is complete w.r.t. trace pre-order.

The proof of this result is omitted due to the lack of the space and will appear in the full version of this work.

This proposition indicates that a desired test suite can be generated by finding a transition set $\mathcal{R}$ and a distinguishing sequence D such that $\mathcal{R}$ covers T and D is a distinguishing sequence over *states*$(\mathcal{R})$. In the next section, we will show how to find $\mathcal{R}$ and D with a model checker.

5 Test Generation Using Model Checking Tools

Model checking tools such as SPIN [12], SMV [13], UPPAAL [14] are originally designed to verify the correctness of design specifications. Recent years have seen trends in applying model checking tools to assist the test generation procedures (see e.g. [2, 15–19]). When we use a model checker to verify a system model against some required property, a counter-example will be returned if the system model is not correct w.r.t. the property being checked. Making use of this functionality of model checkers, we can characterize a desired test sequence as a property. We use a model checker to verify the negation of this property, called *trap property*, against a system specification. When this trap property is violated, a counter-example returned by the model checker actually serves as a desired test sequence. Following this line of research, we present here another example of using model checkers to generate test sequences in conformance testing with context.

To avoid constructing synchronous product of $\mathcal{S}$ and $\mathcal{C}$, the specifications of the IUT and its context are given to a model checker as a system specification. The specification FSM of the IUT can be straightforwardly translated into any formal specification language accepted model checking tools. For its context, we do not restrict it to be given in a particular specification language or a particular model, as long as it can be translated into a specification language accepted by the adopted model checker. In the following, we use *Spec* to denote the specification for the composition of the IUT and its context given in the specification language of the chosen model checker.

We explain below how to make use of the specification FSM of an IUT and a model checker (with *Spec*) to derive a test suite of the IUT and its context that is complete with respect to trace pre-order.

5.1 Finding Transitions in $\mathcal{R}$

As we explained in Section 4, we need to find a subset $\mathcal{R}$ of transitions in $\mathcal{S} \times \mathcal{C}$ such that $\mathcal{R}$ covers T where T is the set of transitions in $\mathcal{S} \times \mathcal{C}$. Since the synchronous product FSM for the IUT and its context is not available, we analyze $\mathcal{S}$ and derive $\mathcal{R}$ via a model checker. Fig. 1 shows an algorithm to use a model checker to determine a transition set $\mathcal{R}$ such that $\mathcal{R}$ covers T.

A path $\rho = (s_1, s_2, i_1/o_1) \circ (s_2, s_3, i_2/o_2) \circ \ldots \circ (s_k, s_{k+1}, i_k/o_k)$ in $\mathcal{S}$ is *composable* if $i_1 \in I$, $i_j \in X$ for $2 \leq j \leq k$, and $\delta(s_{k+1}, i) \neq \mathit{null}$ for some $i \in I$. According to the definition of synchronous product in Section 3, any transition $t = ((s, c), (s', c'), i/o) \in T$ is constructed from some composable path. On the other hand, not all composable paths in $\mathcal{S}$ can be used to define a transition in $\mathcal{S} \times \mathcal{C}$. Those that can be used to define a transition in $\mathcal{S} \times \mathcal{C}$ are called *executable paths*. Recall that transitions of T in partition $G(s, s', i, o)$ share the same local state s of the IUT in its starting state, the same local state s' of the IUT in its ending state, and the same input i and output o. Each executable path is actually uniquely used to define all transitions in one of the partitions.

Now, as we want to derive a set $\mathcal{R}$ of transitions that contains at least one (arbitrary) transition in each partition, we can use an executable path ρ in $\mathcal{S}$ to

```
1: Input: $\mathcal{S}$, Spec.
2: Output: a set $V$ of pairs of transitions in $\mathcal{S} \times \mathcal{C}$ and input sequences in $I^*$, $\mathcal{R}$.
3: Let $\Phi$ contains all composable paths in $\mathcal{S}$;
4: Let $V = \emptyset$;
5: for each path $\rho$ in $\Phi$ do
6:     define a formula $\phi$ to express the non-existence of a path in Spec which contains a
       subpath which is equal to $\rho$ when all its transitions from the context are ignored.
7:     use model checker to verify formula $\phi$ in Spec;
8:     if formula $\phi$ is violated then
9:         add $(t, \sigma)$ to $V$, where (i) $t \in \mathcal{S} \times \mathcal{C}$ is a transition derived by $\rho$ and a path in
           $\mathcal{C}$ defined by the counter-example returned from the model checker; and (ii)
           $\sigma$ is an input sequence in $I^*$ derived from the counter-example that defines a
           path from $(s_0, c_0)$ to the starting state of $t$;
10:    end if
11: end for
12: Let $\mathcal{R} = \{t \mid (t, \sigma) \in V\}$;
13: return $V$ and $\mathcal{R}$;
```

Fig. 1. Algorithm 1. To find a transition set $\mathcal{R}$

request the model checker to find an arbitrary transition of T that represents the partition uniquely determined by ρ. This can be done as follows: Use temporal logic formula to express such a property that there exists a subpath which is equal to ρ when all its transitions from the context are ignored. Request the model checker to verify the trap property, i.e. the negation of the above property. If ρ is used to define a transition t in $\mathcal{S} \times \mathcal{C}$, then the model checker will detect the violation of the trap property, returning a path in *Spec* from which we can derive a transition in the partition of ρ. Note that in addition to the transition in T, we also derive from the counter-example an input sequence in I^* which defines a path from (s_0, c_0) to the starting state of t. This input sequence will be used later on to construct a test suite.

As statically we do not know which composable path is executable, we simply ask the model checker to check all composable paths. If a composable path is not executable, the model checker will prove the trap property. In this case, we do not need to record any information.

Since $\mathcal{S}$ is finite and free from internal-port-cycles, the number of composable paths in $\mathcal{S}$ is finite and the computation of Φ is in polynomial time. Consequently, the time complexity of Algorithm 1 depends on that of the model checking algorithms used by the model checker. See e.g. [13] for the discussions on the complexity of model checking algorithms. In fact, optimization techniques of model checking have been well studied in recent years to enhance its applicability. Thus, the practicality of Algorithm 1 is endorsed.

According to Algorithm 1, we have the following result. Again, its proof is omitted due to the lack of the space and will appear in the full version of this work.

Proposition 2. *Let T be the set of transitions in $\mathcal{S} \times \mathcal{C}$, and $\mathcal{R}$ the set of transitions obtained from Algorithm 1. We have $\mathcal{R}$ covers T.*

5.2 Finding a Distinguishing Sequence

Algorithms for finding a distinguishing sequence of an FSM are well-discussed in the literature. See [11] for a good survey on this topic. However, finding a distinguishing sequence of an FSM in context is much more complicated. Due to the fact that a distinguishing sequence on $\mathcal{S}$ over $states(\mathcal{R})$ must be calculated with both the specification of the IUT and that of its context, while synchronous product FSM of them is not available, we will apply model checker again. In [20], the authors presented an approach to generating a distinguishing sequence of an EFSM with UPPAAL model checker [14]. Here, we adopt the idea of this approach to generate a distinguishing sequence on $\mathcal{S}$ over $states(\mathcal{R})$.

```
1: Input: $Spec$, $\mathcal{R}$.
2: Output: a distinguishing sequence on $\mathcal{S}$ over $states(\mathcal{R})$.
3: for each state $(s, c)$ in $states(\mathcal{R})$ do
4:    create a variant of $Spec$ with $(s, c)$ as its initial state;
5: end for
6: create a monitor process to synchronize all variants in the sense that a variant can
   only accept an input if all others accept the same input simultaneously;
7: define a formula $\phi$ to express the property that there does not exist an input se-
   quence such that the corresponding output sequences produced by any two variants
   with different local states of $\mathcal{S}$ as their initial states are all different;
8: request model checker to verify $\phi$ in $Spec$;
9: if model check detects a violation then
10:    Let $D$ be the input sequence derived from the counter-example returned by the
       model checker;
11:    return $D$;
12: else
13:    return "There does not exist any distinguishing sequence on $\mathcal{S}$ over $states(\mathcal{R})$";
14: end if
```

Fig. 2. Algorithm 2. To find a distinguishing sequence over $states(\mathcal{R})$

Fig. 2 shows an algorithm for this purpose. Initially, for each state $(s, c) \in states(\mathcal{R})$, we create a variant of $\mathcal{S}$ with s as its initial state and a variant of $\mathcal{C}$ with c as its initial state. Then by making use of a special *monitor* process, we request all the processes that represent these variants of $\mathcal{S}$ to synchronize all their actions on accepting input from both the environment port and the context port so that they will always accept the same input at the same time. For any two variants whose local states of $\mathcal{S}$ in their initial states are different, if the output sequences produced upon a same input sequence are all different, then the input sequence can be used as a desired distinguishing sequence D on $\mathcal{S}$ over $states(\mathcal{R})$.

As we know, not every FSM has a distinguishing sequence, In our setting, we cannot guarantee either their existence. However, as distinguishing sequences very often exist in real-life examples, the distinguishing sequences in our setting also exist in many application examples.

The problem of finding a distinguishing sequence is PSPACE-hard by itself [11]. Algorithm 2 reduces the problem to an application of model checking tools. This allows us to benefit from important features that they provide, such as the efficient partial order reduction and OBDD, and thus, reduce the actual cost for the computation.

Finally, with V and D, a test suite $\mathcal{T}$ is obtained: For each $(t, \sigma) \in V$, add both $\sigma \circ D$ and $\sigma \circ i \circ D$ to $\mathcal{T}$, where i is the input of t.

6 An Application

In this section, we use Inter-library Loan System (ILS) as a running example and we use SPIN [21] as a supporting model checker to show how to use the proposed technique to generate a complete test suite w.r.t. trace pre-order for testing in context.

SPIN targets the efficient verification of a system model against the required properties on-the-fly. Here, the system model is described in Promela [21] and the required system properties are often expressed in Linear Temporal Logic (LTL) formulas. As a matter of fact, a design specification expressed in many other specification languages such as FSM and EFSM can be easily translated into a Promela model.

A simplified ILS consists of two components: a borrowing library and a lending library. A user at the borrowing library can search a book in the lending library. When a book is found, the user can choose either to purchase the book or to issue a loan request. The lending library will always grant the purchase of the book; however, the allowance of the loan of the book depends both on the availability of the required book and on the length of the waiting list. There are three cases: i) if the book is available, the loan request will be granted; ii) if the book is unavailable but the waiting list is not full, the lending library will ask the user if he/she wants to make a reservation; and iii) if the waiting list is full, the lending library will tell the user that the book is unavailable.

Suppose that the borrowing library is the IUT and the lending library is its context. The specification $\mathcal{S}$ of the IUT has two ports: $portUser$ and $portContext$. Port $portUser$ represents the interface of the borrowing library with the environment/tester, and port $portContext$ represents the interface of the borrowing library with its context, the lending library. The semantics of service primitives used in ILS can be inferred by their symbolic representations. For example, $searchBook$ is an input primitives at $portUser$ to represent a user's action of searching a book; $loanAccptd$ is an input primitives at $portContext$ to represent that a user's request of a book loan is accepted.

Fig. 3 and Fig. 4 give the specification FSM $\mathcal{S}$ of the borrowing library and the Promela model of the lending library $\mathcal{C}$, respectively. Suppose that the number of available books is 3, and the length of the waiting list for a book reservation cannot exceed 3. Let T be the set of transitions in $\mathcal{S} \times \mathcal{C}$, and $\mathcal{R} \subseteq T$. To find $\mathcal{R}$ such that $\mathcal{R}$ *covers* T and to find a distinguishing sequence over $states(\mathcal{R})$, we need to translate FSM $\mathcal{S}$ and the behavior of a user of the ILS into Promela processes. Thus, there are three processes in the Promela model of ILS: $User$, $Borrower$ and $Lender$, which represent the specifications of the environment/tester, the borrowing library, and the lending library, respectively. To establish the communication among these processes, there are four channels.

- $fromUser$: a channel through which $Borrower$ receives inputs from $User$;
- $ToUser$: a channel through which $Borrower$ sends outputs to $User$;
- $fromLender$: a channel through which $Borrower$ receives inputs from $Lender$;
- $ToLender$: a channel through which $Borrower$ sends outputs to $Lender$;

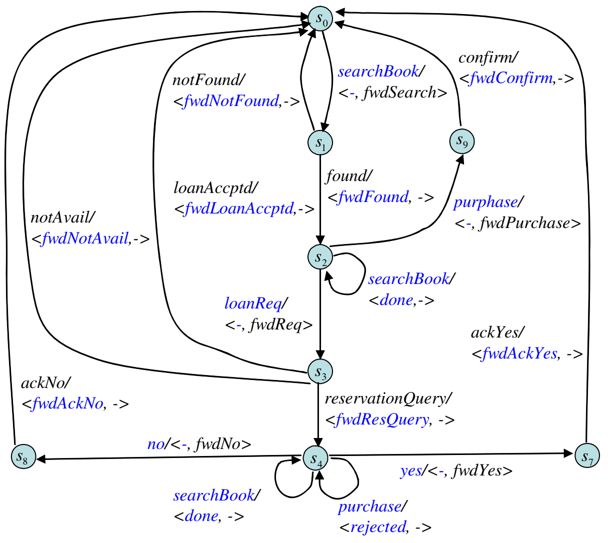

Fig. 3. Specification FSM of the borrowing library

Now we show how to find $\mathcal{R}$. Let $\rho = loanReq/\langle -, fwdReq\rangle \circ notAvail/\langle fwdNotAvail, -\rangle$. Clearly, ρ is a composable path in $\mathcal{S}$. In order to use SPIN to check whether ρ is executable, we need an LTL formula to express the negation of the existence of a transition in $\mathcal{S} \times \mathcal{C}$ derived from ρ.

Since the sending actions are always executable, we focus on finding a path to enable the receiving actions in ρ. Let the temporal logic variables be defined as follows:

$$r = Borrower@s_2$$

$$p = fromUser?[loanReq]$$

$$q = fromLender?[notAvail]$$

Here, r represents that process $Borrower$ is in state s_2; p represents that message $loanReq$ is received from channel $fromUser$; and q represents that message

```
proctype Lender() {
bool book; /*initialization*/
int inStock =3; /*No. of available books*/
int waitingLst = 0;
int Max = 3; /*the maximum length of waiting list*/

if
:: book = true;
:: book = false;
fi;

ac0: /*label ac0 is associated with abstract state ac0*/
if
:: book == true → toLender ? fwdSearch
                          → fromLender ! found;
:: book == false → toLender ? fwdSearch
                          → fromLender ! notFound;
                          goto ac0;
fi;

ac1: /*label ac1 is associated with abstract state ac1*/
if
:: toLender ? fwdReq;
if
:: inStock > 0 → fromLender ! loanAccptd;
                          inStock–;
                          goto ac0;
:: inStock <= 0 and waitingLst >= Max
                          → fromLender ! notAvail;
                          goto ac0;
:: inStock <= 0 and waitingLst < Max
                          → fromLender ! reservationQuery;
fi;
:: toLender ? fwdPurchase → fromLender ! confirm;
                          goto ac0;
fi;

ac2: /*label ac2 is associated with abstract state ac2*/
if
:: toLender ? fwdYes → fromLender ! ackYes;
                          waitingLst++;
                          goto ac0;
:: toLender ? fwdNo → fromLender ! ackNo;
                          goto ac0;
fi;
}
```

Fig. 4. Promela model of the lending library

notAvail is received from channel $fromLender$. Then the desired trap LTL formula can be expressed as

$$\phi =!(<> (rUp)Uq).$$

When verifying the ILS Promela model against ϕ, we obtain the following result from the returned counter-example:

$\sigma = searchBook \circ loanReq \circ searchBook \circ loanReq \circ searchBook \circ loanReq \circ searchBook \circ loanReq \circ yes \circ searchBook \circ loanReq \circ yes \circ searchBook \circ loanReq \circ yes \circ searchBook$

$t = ((s_2, c_{1,2}), (s_0, c_{0,4}), loanReq/\langle -, fwdReq\rangle \circ notAvail/\langle fwdNotAvail, -\rangle)$,

where $c_{0,4}$ and $c_{1,2}$ are concrete states split from abstract state ac_0 and ac_1 in the situation when $inStock = 0$ and $waitingLst = 3$, respectively.

This result actually describes a possible scenario of having a transition in $\mathcal{S} \times \mathcal{C}$ derived from ρ when all the books in the lending library are checked out and the waiting list is full.

As shown in [11], the role of distinguishing sequences can actually be replaced by their prefixes, one for each state. This very often helps us achieve shorter test sequences. The definition of a distinguishing sequence over $\mathcal{W}$ can be extended to prefix distinguishing sequences D_i (for state s_i) straightforwardly. Following Algorithm 2, prefix distinguishing sequence D_i over $states(\mathcal{R})$ can be found with SPIN. For example, we have $D_0 = searchBook$ and $D_2 = D_4 = searchBook \circ purchase$. Thus, test sequences for t are $\sigma \circ D_2$ and $\sigma \circ loanReq \circ D_0$.

7 Related Work

There are various types of applications of using a model checker to generate tests. Ammann et al. combined model checking with *mutation analysis* to generate test cases [22]: after a specification model is mutated by applying mutation operators, a model checker generates counter-examples to distinguish the mutant models from the original specification model, and thus test cases are derived. Gargantini and Heitmeyer presented a technique to construct test sequences upon a special class of so-called *Software Cost Reduction* requirements, by using a model checker [23]. In order to save memory from a huge predefined test suite, Tretmans and de Vries [24] used model checker SPIN to generate tests *during testing* for non-deterministic stateful systems. How to generate test cases according to some *data flow test selection criteria* is discussed in [25]. In [20], Goltz et al. used a model checker to generate a shortest distinguishing sequence of an EFSM. In terms of applying model checking tools for test generation, we have added one more example along this line of research, particularly for testing in context.

Along the approaches of testing in context, there are several possible ways to interpret the context of an IUT. Petrenko et al. considered the situation where the IUT is an embedded component and its communication with the environment has to be carried out through its context. For this case, they presented a framework of testing an embedded component in context [2, 16]. In particular, the problems of test executability and fault propagation are addressed in the presence of the context. In [15, 17–19], different approaches are discussed for solving the problem of translating internal tests derived for an embedded component into external observable tests of the entire system. Different from their test architecture, our work is applicable to testing an IUT that is associated with an embedded component.

8 Conclusion and Final Remarks

In this paper, we presented a method of deriving a *complete* test suite w.r.t. trace pre-order for testing the IUT with an embedded context, and provided a way of implementing this method by making use of model check tools.

As an initial piece of work on testing in context with model checkers, our focus has been put on the general method. Further improvements can be made in terms of the size of the constructed test suite. For example, it is possible to reduce the size of the generated test suite by constructing a *test tree* similar to the one introduced in [3]; We can adopt those model checkers that can always find *shortest* counter-examples in terms of the lengths so that shorter test sequences can be derived. Apart from the optimization issue, there are many other directions to extend our current work.

– It remains interesting to discuss our test generation technique in more general situations where both the IUT and its context have communications with the environment.
– IUT may be nondeterministic: we would like to study how to extend our results to nondeterministic testing in context.
– When the IUT is completely specified, it is not always possible to achieve trace equivalence due to the interoperability of the IUT and its context. We would like to discuss the condition on $\mathcal{S}$ and $\mathcal{C}$ such that trace equivalence can be achieved.
– We have used distinguishing sequence for state identification. At expense of its convenience for testing, distinguishing sequence does not always exist. Although the use of characterization set usually results in much bigger test suites, a characterization set is more likely to exist in an FSM with context. Therefore, we would like to study on how to use model checking tools to generate characterization set in our setting.

Acknowledgments. This work is supported by the Natural Sciences and Engineering Research Council of Canada under grant number RGPIN 209774.

References

1. Tretmans, J.: Conformance testing with labelled transition systems: Implementation relation and test generation. Computer Networks and ISDN Systems 29, 49–79 (1996)

2. Petrenko, A., Yevtushenko, N., von Bochmann, G., Dssouli, R.: Testing in context: framework and test derivation. Computer Communications 19(14), 1236–1249 (1996)
3. Chow, T.: Testing software design modeled by finite-state machines. IEEE Trans. Software Eng. SE-4(3), 178–187 (1978)
4. Sabnani, K., Dahbura, A.: A protocol test generation procedure. Computer Networks and ISDN Systems 4(15), 285–297 (1988)
5. Hennie, F.: Fault detecting experiments for sequential circuits. In: Proc. of 5th Ann. Symp. Switching Circuit Theory and Logical Design, pp. 95–110 (1964)
6. van der Bijl, M., Rensink, A., Tretmans, J.: Compositional testing with ioco. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, pp. 86–100. Springer, Heidelberg (2004)
7. Hierons, R.M., Ural, H.: UIO sequence based checking sequences for distributed test architectures. Information and Software Technology 45(12), 793–803 (2003)
8. Chen, J., Hierons, R.M., Ural, H., Yenigun, H.: Eliminating redundant tests in a checking sequence. In: Khendek, F., Dssouli, R. (eds.) TestCom 2005. LNCS, vol. 3502, pp. 146–158. Springer, Heidelberg (2005)
9. Gonenc, G.: A method for the design of fault detection experiments. IEEE Trans. Computers 19(6), 551–558 (1970)
10. Ural, H., Wu, X., Zhang, F.: On minimizing the lengths of checking sequences. IEEE Transactions on Computers 46(1), 93–99 (1997)
11. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines — a survey. Proceedings of The IEEE 84(8), 1090–1123 (1996)
12. Holzmann, G.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
13. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
14. Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W.: UPPAAL - a tool suite for automatic verification of real-time systems. In: Proc. of the DIMACS/SYCON workshop on Hybrid systems III: verification and control: verification and control, pp. 232–243 (1995)
15. Lima, L.P., Cavalli, A.R.: A progmatic approach to generating test sequences for embedded systems. In: Proc. of 10th International Workshop on Testing of Communicating Systems, pp. 125–140 (1997)
16. Petrenko, A., Yevtushenko, N., von Bochmann, G.: Fault models for testing in context. In: Proc. of Internation Conference on Formal Techniques for Networked and Distributed Systems, pp. 125–140 (1996)
17. Petrenko, A., Yevtushenko, N.: Testing faults in embedded components. In: Proc. of 10th International Workshop on Testing of Communicating Systems, pp. 272–287 (1997)
18. El-Fakih, K., Petrenko, A., Yevtushenko, N.: FSM test translation through context. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 245–258. Springer, Heidelberg (2006)
19. El-Fakih, K., Yevtushenko, N.: Fault propagation by equation solving. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 185–198. Springer, Heidelberg (2004)
20. Robinson-Mallett, C., Liggesmeyer, P., Mcke, T., Goltz, U.: Generating optimal distinguishing sequences with a model checker. ACM SIGSOFT Software Engineering Notes 30(4), 1–7 (2005)
21. Holzmann, G.: The Design and Validation of Computer Protocols. Prentice-Hall, Englewood Cliffs (1991)

22. Ammann, P.E., Black, P.E., Majurski, W.: Using model checking to generate test from specifications. In: Proc. of 2nd IEEE International Conference on Formal Engineering Methods (ICFEM 1998), pp. 46–54 (1998)
23. Gargantini, A., Heitmeyer, C.: Using model checking to generate tests from requirements specifications. ACM SIGSOFT Software Engineering Notes 24(6), 146–162 (1999)
24. de Vries, R., Tretmans, J.: On-the-fly conformance testing using Spin. International Journal on Software Tools for Technology Transfer 2(4), 382–393 (2000)
25. Hong, H.S., Lee, I., Sokolsky, O., Ural, H.: Data flow testing as model checking. In: Proc. of IEEE ICSE 2003, pp. 232–242 (2003)