Specification Coverage Aided Test Selection

Tuomo Pyhälä
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O. Box 5400, FIN-02015 HUT, Finland
Tuomo.Pyhala@hut.fi

Keijo Heljanko
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O. Box 5400, FIN-02015 HUT, Finland
Keijo.Heljanko@hut.fi

Abstract

In this paper test selection strategies in formal conformance testing are considered. As the testing conformance relation we use the **ioco** *relation, and extend the previously presented on-the-fly test generation algorithms for* **ioco** *to include test selection heuristic based on a specification coverage metric. The proposed method combines a greedy test selection with randomization to guarantee completeness. As a novel implementation technique we employ bounded model checking for lookahead in greedy test selection.*

1 Introduction

Conformance testing aims to show that a particular implementation conforms to its specification. It is particularly useful in testing implementations of communication protocols. In tele- and data communication field the products of different vendors must be able to cooperate, and this can achieved by conforming to a common specification.

Formal conformance testing formalizes the concepts of conformance testing [16]. Essential notions include the implementation, the specification and conformance relation between these two. In this work we use **ioco** conformance relation [17]. Previously there has been work implementing on-the-fly-testers [4] for **ioco** conformance relation. These basically test whether the implementation behaves according to the specification. The tester has to make choices: what inputs it gives to the implementation and when it waits for an output. How to make these choices well is the problem we concentrate on. All possible test cases cannot be executed, as there often are infinitely many of them. Making good choices reduces the amount of executed test events or raises the quality of testing. We approach this problem with test selection methods based on coverage.

In this paper we introduce a method for using specification coverage to guide test selection by extending the existing on-the-fly algorithm of [4] to include coverage based test selection heuristic while still preserving the completeness of procedure. (See [7] for another approach to introduce probabilities to the same algorithm.) We have selected a specification based coverage metric and implemented an on-the-fly testing framework employing this method. Initial experiments to evaluate the efficiency of the suggested approach are presented. We are able to demonstrate cases where the suggested approach is very efficient. Experiments on a more real life case study, the conference protocol, demonstrate that the method is feasible, however, the achieved improvements are not as dramatic.

Coverage has been used in software testing [1], e.g., statement, branch and path coverage, and we try to apply similar ideas to formal conformance testing. Using coverage one might for example wish to execute all the lines of source code at least once. Unfortunately, in black box testing this is not possible, because we do not know the internals of the actual implementation. Instead, we assume that the implementation resembles the specification and try to cover the specification. From a pragmatic point of view, if the implementation is made according to the specification (or vice versa) it is somewhat likely that they resemble each other. Therefore we take the assumption that in many cases arising in practical test settings, specification based coverage can "approximate" coverage used in white box testing.

Existing work has investigated selecting tests and measuring coverage based on traces [8], test selection using test purposes [3], using coverage information to find bugs faster in model checking Java programs [9], using heuristic methods for test selection [14], using bounded model checking in test sequence generation [18], and using coverage criteria in test generation for white box testing [15].

We assume that the sending and receiving testing events from the implementation is the most time consuming task. For example, test generation is considered to take a lot less time than executing the tests. Therefore our target is to reduce the amount executed test events while preserving the quality of test suite, e.g., the capability to detect errors. The test selection heuristic combines a greedy algorithm

with randomization. As a novel implementation technique we use bounded model checking [2] for lookahead in the greedy part of the heuristic. Bounded model checking is an interesting new technology which enables new algorithms also in the area of testing. We believe that the suggested method is a very interesting application area for bounded model checking, as trying to navigate in a non-deterministic specification to an uncovered part of the specification can be seen as a model checking subroutine. Furthermore, failing to find a counterexample (for example due to a user imposed time limit on the bounded model checker) will only result in the test selection algorithm falling back to randomized test selection strategy, which only degrades the test selection performance, but does not affect the soundness or completeness of the procedure. As an alternative to bounded model checking also some random walk lgorithms could be used instead. However, their performance on some specifications, like the combination lock example in Sect. 8, will not be as good as those obtained using a bounded model checker.

The main contribution of the paper is a new test selection method based on specification coverage. The algorithm can be seen as an extension of the algorithm of [4]. We have implemented the method, and demonstrate the feasibility of the approach using a small set of experiments. The implementation uses bounded model checking as a novel implementation technique.

The structure of the the rest of this paper is as follows. In Sect. 2 we introduce the labelled transition systems (LTS's for short) and in Sect. 3 we introduce the conformance relation **ioco** for LTS's. Section 4 describes the on-the-fly testing and Sect. 5 introduces our specification formalism, labelled 1-safe Petri nets, and the coverage metric we use. Section 6 contains the new algorithms for on-the-fly testing. Section 7 introduces the conference protocol case study, Sect. 8 describes the experiments made, and finally Sect. 9 contains the conclusions.

2 Labelled Transition Systems

Labelled transition systems (abbreviated as LTS) are a well known formalism to specify system behavior. The **ioco** conformance relation has been introduced using LTS's, and we next will introduce the needed notation.

Definition 1 *A* labelled transition system *is a four tuple* (S, L, Δ, s_0)*, where* S *is set of states,* L *is a finite set of* labels *with a special symbol* $\tau \notin L$*,* $\Delta \subseteq S \times (L \cup \{\tau\}) \times S$ *is the transition relation, and* s_0 *is the* initial state*.*

We will not always distinguish between an LTS and its initial state, and for convenience often use the initial state to also denote the LTS itself. When testing systems we have visible actions and hidden internal actions. We define the set of visible actions as follows.

Definition 2 *The set of visible transitions of an LTS* $p = (S, L, \Delta, s_0)$ *is* $\Delta_v =_{def} \{(s, a, s') \in \Delta \mid a \in L\}$.

We use notation L^*, where L is a finite set of labels, to denote the set of finite sequences over L. The following notation is been defined to be identical with [4]

Definition 3 *Let* $p = (S, L, \Delta, s_0)$ *be an LTS. We define the following notation, where* $s, s' \in S$*,* $S' \subseteq S$*,* $\mu, \mu_i \in L \cup \{\tau\}$*,* $a, a_i \in L$*:*

$$
\begin{array}{rcl}
s \xrightarrow{\mu} s' & =_{def} & (s, \mu, s') \in \Delta, \\
s \xrightarrow{\mu_1 \ldots \mu_n} s' & =_{def} & \exists s_1, s_2, \ldots, s_{n-1} : \\
 & & s \xrightarrow{\mu_1} s_1 \cdots s_{n-1} \xrightarrow{\mu_n} s', \\
s \xrightarrow{\mu_1 \ldots \mu_n} & =_{def} & \exists s' : s \xrightarrow{\mu_1 \ldots \mu_n} s', \\
s \xrightarrow{\mu_1 \ldots \mu_n} \not & =_{def} & \neg s \xrightarrow{\mu_1 \ldots \mu_n}, \\
s \stackrel{\epsilon}{\Longrightarrow} s' & =_{def} & s = s' \vee s \xrightarrow{\tau \ldots \tau} s', \\
s \stackrel{a}{\Longrightarrow} s' & =_{def} & \exists s_1, s_2 : \\
 & & s \stackrel{\epsilon}{\Longrightarrow} s_1 \xrightarrow{a} s_2 \stackrel{\epsilon}{\Longrightarrow} s', \\
s \stackrel{a_1 \ldots a_n}{\Longrightarrow} s' & =_{def} & \exists s_1 \ldots s_{n-1} : \\
 & & s \stackrel{a_1}{\Longrightarrow} s_1 \cdots s_{n-1} \stackrel{a_n}{\Longrightarrow} s', \\
s \stackrel{\sigma}{\Longrightarrow} & =_{def} & \exists s' : s \stackrel{\sigma}{\Longrightarrow} s', \\
s \stackrel{\sigma}{\Longrightarrow} \not & =_{def} & \neg s \stackrel{\sigma}{\Longrightarrow}, \\
\mathit{traces}(s) & =_{def} & \{\sigma \in L^* \mid s \stackrel{\sigma}{\Longrightarrow}\}, \\
\mathit{init}(s) & =_{def} & \{\mu \in L \cup \{\tau\} \mid s \xrightarrow{\mu}\}, \\
s \textbf{ after } \sigma & =_{def} & \{s' \mid s \stackrel{\sigma}{\Longrightarrow} s'\}, \\
\mathit{init}(S') & =_{def} & \bigcup_{s \in S'} \mathit{init}(s), \text{ and} \\
S' \textbf{ after } \sigma & =_{def} & \bigcup_{s \in S'} s \textbf{ after } \sigma.
\end{array}
$$

Definition 4 *A divergence free (i.e, strongly convergent) LTS does not contain an infinite execution of* τ *transitions, i.e.,* $\exists n \in \mathbb{N} : \forall s_1, s_2, \ldots, s_n$ *: if* $s_1 \xrightarrow{\tau} s_2 \cdots s_{n-1} \xrightarrow{\tau} s_n$ *then it follows that* $s_n \xrightarrow{\tau} \not$ s_{n+1}*.*

We make the same restrictions of LTS's as [4]. We restrict ourselves to divergence free labelled transition systems, which are denoted as $\mathcal{LTS}$. To distinguish between inputs and output, we define a subclass of labelled transition systems called *input-output transition systems* to represent the implementation under test. We also require that IOTS must always accept all inputs. The IOTS are very closely related to I/O Automata of Lynch et. al, see e.g., [4].

Definition 5 *An* input-output transition system *(abbreviated as IOTS) is an LTS with the following restrictions. The set of labels L is divided into input labels L_I and output labels L_U, such that $L = L_I \cup L_U$ and $L_I \cap L_U = \emptyset$. Furthermore for an IOTS it is required that $\forall s \in S : \forall a \in L_I : s \stackrel{a}{\Longrightarrow}$. We denote the set of all input-output transition systems with $\mathcal{IOTS}$.*

3 Conformance Relation ioco

We define **ioco** for labelled transition systems as in [4]. Often system under testing (SUT) has states, where no output can be observed. To model such behaviour with the theory, the *quiescence* concept is introduced.

Definition 6 *A state s of $i \in \mathcal{LTS}$ (with L_U and L_I defined as for IOTS) is quiescent iff $\forall \mu \in L_U \cup \{\tau\} : s \stackrel{\mu}{\not\longrightarrow}$. A quiescent state s is denoted by $\delta(s)$.*

The concept of out sets will be introduced to represent the possible outputs of some particular state.

Definition 7 *Let s be a state of $i \in \mathcal{LTS}$ (with L_U and L_I defined as for IOTS), then*

$$\mathit{out}(s) \quad =_{def} \quad \{a \in L_U \mid s \stackrel{a}{\Longrightarrow}\} \cup \{\delta \mid \delta(s)\}.$$

Suspension traces are the traces including suspensions, i.e., apart from having symbols from the label set L, they also have δ symbols to denote suspensions.

Definition 8 *For this definition extend the $s \stackrel{\mu}{\longrightarrow} s'$ notation. It is defined to include the suspensions in following way: If $\delta(s)$, then $s \stackrel{\delta}{\longrightarrow} s$. After this change $s \stackrel{a}{\Longrightarrow} s'$ as well as other notations in Def. 3 are extended to use the redefined $s \stackrel{\mu}{\longrightarrow} s'$ as a basis for them. The set of suspension traces is defined as follows.*

$$\mathit{Straces}(s) \quad =_{def} \quad \{\sigma \in (L \cup \{\delta\})^* \mid s \stackrel{\sigma}{\Longrightarrow}\}.$$

Now we have introduced required concepts to define the **ioco** conformance relation.

Definition 9 *Let $i \in \mathcal{IOTS}$ be a implementation and $s \in \mathcal{LTS}$ be a specification, then*

$$i \textbf{ ioco } s \quad =_{def} \quad \forall \sigma \in \mathit{Straces}(s) : \mathit{out}(i \textbf{ after } \sigma) \subseteq \mathit{out}(s \textbf{ after } \sigma).$$

In other words, an implementation i will be non-conformant if after a trace of the specification ($\sigma \in$ Straces(s)), it produces an output, say a, which the specification cannot match ($a \notin$ out(s **after** σ)). Do note that non-deterministic specifications are allowed by the formulation above.

4 On-the-fly Testing

In on-the-fly testing the running of a test and the generation of a test case are combined into one algorithm. This is quite useful in practice as often the behavior of the implementation restricts the set of possible tests runs. An efficient on-the-fly testing tool has been presented for the **ioco** conformance relation in [4] based on the model checker SPIN [12]. The starting point of our work was the on-the-fly testing algorithm presented in [4]. Their algorithm for test selection is fully non-deterministic, and we have come up with a coverage based test selection algorithm as an alternative to test selection.

The Algorithm 1 is the main on-the-fly testing routine. This routine is parameterized by the selection of the subroutine *TestMove*(S). If we use the Algorithm 2 (*RandomTestMove*(S)) as the subroutine *TestMove*(S), we basically get a randomized implementation of the on-the-fly testing algorithm of [4]. Later we will replace *TestMove*(S) by another subroutine called *HeuristicTestMove*(S), and the development of this subroutine is the main contribution of this work. If Algorithm 1 terminates the implementation is not an **ioco** conformant one. We do not discuss in this paper how other test runs should be terminated, as those details are similar to [4].

5 Petri Nets

Petri nets [5] are a formalism, which enables a more compact representation of systems, than simple labelled transition systems as introduced in Def. 1. A similar compact representation could also be achieved using synchronizations of such simple labelled transition systems, but for this work we chose labelled 1-safe Petri nets. They are guaranteed to induce LTS's with finite set of reachable states. As we also had a bounded model checker available for 1-safe Petri nets [11], we felt that choosing 1-safe Petri nets for the task would be convenient.

A three tuple $N = (P, T, F)$ is a *net*, where P and T are finite sets of *places* and *transitions*, respectively. The place and transition sets are distinct, i.e., $P \cap T = \emptyset$. The *flow relation* is $F \subseteq (P \times T) \cup (T \times P)$. Transition and places can also be called *nodes*. By $F(x, y)$ we denote the *characteristic function* of F. It is defined as $(x, y) \in F \Rightarrow F(x, y) = 1$ and $(x, y) \notin F \Rightarrow F(x, y) = 0$.

A *marking* of a net (P, T, F) is a function $M : P \rightarrow \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers including zero. Marking associates a number of tokens with each place.

A four tuple $N = (P, T, F, M_0)$ is a *Petri net* if (P, T, F) is a net, and M_0 is a marking of this net. A transition t is *enabled* in a marking M, iff $\forall p : M(p) \geq F(p, t)$. If transition t is enabled in a marking M, it can occur leading to marking M', where $\forall p \in P : M'(p) = M(p) -$

Algorithm 1.

```
procedure OnTheFlyTest(IOTS i, LTS s) {
  terminate := false; failure := false;
  S := {s0} after ε; // Calculate all the initial states
  while (terminate = false) {
    move := TestMove(S); // Returns move ∈ L_I ∪ {output}
    if (move ∈ L_I) then {
      x := move; // We have x ∈ (init(S) ∩ L_I)
      Stimulate(i, x); // Stimulate the implementation
      TestLog(S, x); // Log the input
      S := S after x; // Update spec state set
    } else { // move = output
      x = ObserveOutput(i);
      TestLog(S, x); // Log the output or timeout observed
      if (x ∈ out(S)) then {
        S := S after x; // Update spec state set
      } else {
        failure := true; terminate := true;
      }
    }
  }
  return failure;
}
```

Algorithm 2.

```
procedure RandomTestMove(superstate S) {
  inputs := init(S) ∩ L_I; // Calculate possible inputs
  move := output;
  if (inputs ≠ ∅) then {
    if (TrueWithProbability(Prob_input)) then {
      move := PickRandomElement(inputs); // Pick an input
    }
  }
  return move;
}
```

Figure 1. The on-the-fly testing algorithm main loop and randomized test selection subroutine

$F(p,t) + F(t,p)$. This is denoted by $M \xrightarrow{t} M'$.

In further discussion we will discriminate between *reachable markings* and all markings.

Definition 10 *Let $N = (P, T, F, M_0)$ be a Petri net. The set of reachable markings RM(N) is defined to be the smallest set fulfilling following two conditions:*

(i) $M_0 \in$ *RM*(N)*, and*

(ii) if $M \in$ *RM*(N) *and* $M \xrightarrow{t} M'$ *then* $M' \in$ *RM*(N).

A marking M is reachable, iff $M \in$ *RM*(N).

In 1-safe Petri nets for all markings $M \in \mathrm{RM}(N)$ it holds that $\forall p \in P : M(p) \leq 1$. We will restrict ourselves to 1-safe Petri nets in this work. To make a connection between Petri nets and LTS's we define a labelling on the transitions of the Petri net.

Definition 11 *A* labelled net *is a four tuple (P, T, F, λ), where (P, T, F) is a net, and $\lambda : T \to L \cup \{\tau\}$ attaches labels to transitions from a finite set of labels L.*

To use our **ioco** conformance relation defined for LTS's, we have to define how an LTS is formed on the basis of a Petri net. We include the reachable markings as the states of such LTS and if some transition exists between the states then we include an arc with a corresponding label to the LTS. We define an induced LTS of a labelled Petri net as follows

Definition 12 *A labelled Petri net $N = (P, T, F, \lambda, M_0)$ induces a labelled transition system defined as LTS$(N) = (S, L, \Delta, s_0)$, where $S =$ RM(N), L is the set of labels in the labelled Petri net, $\Delta = \{(M, l, M') \mid M \xrightarrow{t} M' \wedge \lambda(t) = l\}$, and $s_0 = M_0$.*

As a side note, we remind the reader we have extended the notation of $s \xrightarrow{\mu} s'$ to include the suspension arcs $s \xrightarrow{\delta} s$ for all LTS, including LTS(N). We assume in the rest of this work that LTS(N) fulfills all the restrictions put on LTS's in the previous section.

In testing context it is important to discriminate between visible actions and internal actions. The visible actions are those in the set L, and the only internal action is the label τ.

Definition 13 Visible transitions *of labelled Petri net $N = (P, T, F, \lambda)$ are transitions $t \in T$, such that $\lambda(t) \neq \tau$. We denote with vistrans(N) the set of visible transitions of N.*

Assume that during on-the-fly testing we have run a test sequence σ. We want to define a coverage metric that records which visible transitions of the net have been covered during the test run. We require that the coverage metric should be monotone, in the sense that a longer test sequence always covers at least as many visible transitions as a shorter one.

Definition 14 *Let N be a specification Petri net, and let LTS(N) be the induced LTS. A test run $\sigma \in (L \cup \{\delta\})^*$ covers the transition $t \in$ vistrans(N), iff there exists $\sigma', \sigma'' \in (L \cup \{\delta\})^*$, $s \in S$, and $M, M' \in$ RM(N) such that:*

(i) $\sigma = \sigma' \cdot \lambda(t) \cdot \sigma''$,

(ii) $s_0 \xRightarrow{\sigma'} s = M$, *and*

(iii) $M \xrightarrow{t} M'$.

The intuition behind the visible net transition coverage is the following. After the test sequence σ' our on-the-fly testing algorithm has computed a superstate S which contains the state s, corresponding to the marking M of the net. When we at that point continue testing by (input or output) $\lambda(t)$, we have to also consider the case where the net transition t is fired when computing S **after** $\lambda(t)$, and at that point the transition t becomes covered.

6 Coverage Based Test Selection

The approach to using specification coverage to guide test selection in this work is very pragmatic. When testing a system we have usually only a finite amount of test resources available. In this work we assume that the number of test moves, in other words inputs sent to the implementation and outputs observed from it, can be used as a measure of the resources used for testing. We would like to minimize the amount of testing needed to detect non-conforming implementations.

When seen from a theoretic perspective, our work might seem futile. If we see testing as a game against a truly non-deterministic black-box implementation with no limitations to its internal structure, using a fully non-deterministic test selection strategy will surely be the best we can do from a theoretical perspective.

However, from a pragmatic point of view, often the test setting is far from the worst case scenario. The internal structure of the implementation can be similar to the internal structure of the specification, and the implementation might be deterministic or randomized (instead of playing against us). This gives us hope that increasing coverage on the specification side during testing will also lead to increasing coverage on the implementation side, and more importantly to finding non-conforming implementations more effectively. Again, from a pragmatic point of view, only practical experimentation will show whether this hope in using specification based coverage metrics for testing is unfounded.

6.1 Coverage Based Test Selection Heuristic

The on-the-fly testing main loop in Algorithm 1 is kept as is when incorporating coverage based heuristics, as it basically just executes test moves provided by the *TestMove*(S) subroutine. The first extension is to keep track of the used coverage metric. The subroutine *TestLog*(S, x) can be extended to keep track of the coverage metric in use. In the experiments presented in this work we use visible transition coverage of the specification Petri net (see Def. 14), but any other specification coverage metric could alternatively be used. The second change is to use the Algorithm 3 (*HeuristicTestMove*(S)) as the subroutine *TestMove*(S). It will with probability $Prob\_greedy$ (which should always be < 1) call a greedy coverage based test selection subroutine, Algorithm 4. With probability $(1 - Prob\_greedy)$, and also in cases the greedy test selection subroutine could not provide anything, we call the already presented random test selection subroutine, Algorithm 2.

The motivation behind Algorithm 4 is as follows. We take an optimistic view of the test setting, in which the implementation plays on our side to increase specification coverage. Then we greedily choose either to fire an uncovered input (guaranteed to increase coverage), or ask for an output (increases coverage only when the implementation gives us an uncovered output, which we optimistically assume it will do for us). If the greedy heuristic could not find a move to take, we employ Algorithm 5 to find a coverage improving test sequence. If such a sequence could be found, we choose the test move based on the first action of that test sequence. In some sense Algorithm 5 can be seen as a generalization of Algorithm 4 for deeper "lookahead" into the set of possible test executions starting from superstate S.

Algorithm 3.

```
procedure HeuristicTestMove(SuperState S) {
  move := none;
  if (TrueWithProbability(Prob_greedy)) then {
    move := GreedyTestMove(S);
  }
  if (move = none) then {
    move := RandomTestMove(S);
  }
  return move;
}
```

Algorithm 4.

```
procedure GreedyTestMove(SuperState S) {
  input_uncovered := UncoveredInputIsEnabled(S);
  output_uncovered := UncoveredOutputIsEnabled(S);
  if (input_uncovered ∧ output_uncovered) then {
    if (TrueWithProbability(Prob_input)) then {
      choice := input;
    } else {
      choice := output;
    }
  } elsif (input_uncovered ∧ ¬output_uncovered) then {
    choice := input;
  } elsif (¬input_uncovered ∧ output_uncovered) then {
    choice := output;
  } else {
    choice := none;
  }
  if (choice = input) then {
    move := PickRandomUncoveredInput(S);
  } elsif (choice = output) then {
    move := output;
  } else {
    move := LookaheadTestMove(S);
  }
  return move;
}
```

Figure 2. Heuristic test selection algorithm and coverage based greedy test selection subroutine for it.

Algorithm 5.

procedure *LookaheadTestMove*(SuperState S) {
  With limited computational resources try to find a (preferably) short execution: $S \stackrel{\sigma}{\Longrightarrow} S'$, such that any test run beginning with σ increases coverage.
  move := $none$;
  if (σ was found) **then** {
    Let x be the first action of σ
    if ($x \in L_I$) **then** {
      move := x;
    } **else** {
      move := $output$;
    }
  }
  return move;
}

Figure 3. An abstract subroutine description for coverage based lookahead

6.2 Implementing Lookahead with Bounded Model Checking

Our implementation of the Algorithm 5 is based on the bounded model checking (BMC) algorithm for 1-safe Petri nets described in [11], incorporating the process semantics optimization described in [10]. Bounded model checking is a recently introduced method [2] for exploring all the possible behaviors of a finite state system (such as a sequential digital circuit or a 1-safe Petri net) up to a fixed bound k. The idea is roughly the following. Given e.g., a sequential digital circuit, a (temporal) property to be verified, and a bound k, the behavior of the sequential circuit is unfolded up to k steps as a Boolean formula S and the negation of the property to be verified is represented as a Boolean formula $\overline{R}$. The translation to Boolean formulae is done so that $S \wedge \overline{R}$ is satisfiable iff the system has a behavior violating the property of length at most k.

In our case the temporal property to be verified says that all executions of length k (of the specification) starting from a state $s_i \in S$, where S is the current superstate of the specification, are such that the last visible event of that execution is either invisible, or a covered transition. Thus a property violation is an execution of length k with a last transition which is both visible and uncovered. Tests beginning with such an execution lead to increasing coverage.

Our implementation tries to find an execution of length k by trying values from 2 to 10. If no execution could be found with bound of 10 we give up, and fall back to fully random test move selection. We try several initial states s_i for each bound k, thus increasing our chances to find a suitable execution. The maximum number of initial states randomly sampled from the superstate S is picked from the sequence $(32, 16, 8, 4, 2, 1, 1, 1, 1)$, e.g., for $k = 3$ we generate 16 different BMC instances.

The Algorithm 5 can in our implementation also be disabled, in which case we call the algorithm "greedy heuristic". When it is enabled, we call it a "BMC heuristic".

7 Conference Protocol

The conference protocol implements a chat service. It enables users to form conferences, and exchange messages with other partners in the conference [6, 4]. The entities of the protocol are depicted in Fig. 4, and are as follows.

Conference service access point (abbreviated CSAP) offers the service primitives: join(nickname, conference), datareq(message), dataind(nickname, message) and leave(). At first user has to join a conference. Once an active member of a conference, user is able to send messages to other members using datareq and receive such messages sent by others using dataind. Once finished conferencing, the user may issue leave() primitive.

Conference protocol entity (abbreviated CPE) uses User Datagram Protocol (abbreviated UDP) as a lower level service to implement the conference service. Each service primitive maps as one or more received or sent UDP protocol data unit (PDU). The CPE has two sets, a preconfigured set of all potential conference partners and a dynamic set of current conference partners.

Issuing a join primitive causes CPE to send all the potential partners a UDP PDU, called join-pdu, containing who is joining to which conference. Existing members of the conference then add the joiner to their dynamic member sets and reply with answer pdu, called answer-pdu. Receiving answer-pdu causes corresponding CPE add the sender of pdu to their conference partner set. Once datareq primitive is issued, CPE sends data-PDU to all partners in the set of current conference partners. Receiving data-PDU causes a dataind primitive to be issued in respective CPE. Receiving leave-PDU causes respective CPE to remove the sender from conference member set. Sending the leave-PDU is caused by leave primitive at CSAP, and it is sent to all the members of the current conference. For more defaults on the case study, see [6, 4].

7.1 Our Model of Conference Protocol

We have created a specification as a high level net, which was converted to a 1-safe Petri net using the Maria tool [13] and then used as an input to our testing tool. The specification is inspired by the Promela specification available in the WWW [6].

Basically our specification model contains no buffering of the inputs. The buffering is not required, because

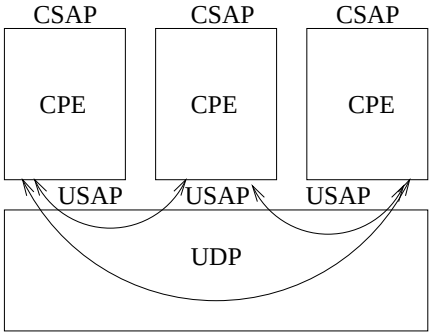

Figure 4. Conference protocol with 3 entities

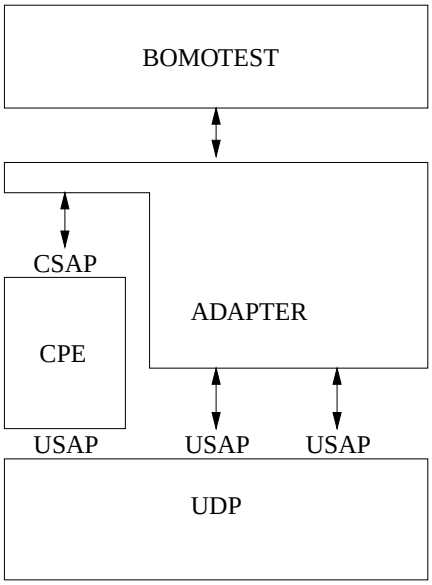

Figure 5. Conference protocol test setup

there could be any input available anyway. The outputs are buffered by having a place, which contains tokens corresponding data packets in transit. The capacity of this place is restricted to hold at most one packet with each (receiver,sender,type)-combination. While we believe that our specification is quite faithful, our limited buffering of outputs makes the specification subtly different from the (Promela) specification of [6]. The chosen modeling of buffers made the specification finite state, and enabled us to use BMC model checker of [11]. The specification is available from `http://www.tcs.hut.fi/~tpyhala/ACSD2003`.

8 Experimental Results

We ran two kinds of experiments: Combinatoric lock and conference protocol. First is chosen to be a example of a case where our coverage metric performs exceptionally

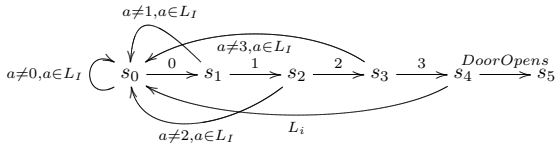

Figure 6. Combinatoric lock LTS

well, and latter is more realistic case where we do not see so dramatic advantages. In both examples we used parameters $Prob\_greedy = 0.75, Prob\_input = 0.5$.

The combinatoric lock example models locks often found in doors, where one is able to press digits and after entering the correct sequence and waiting for a while the door opens. We use a code of length 4 and we have made mutations, where code differs. We wish to detect these mutants using our testing tool. This requires one to enter either the correct code detecting the door staying locked, or to enter the wrong code detecting the door opening with wrong code. An LTS specifying a combinatoric lock with code 0123 is presented in Fig. 6.

The experiments ran show that BMC based heuristic is able to find faulty implementations several decades faster than random walk. The greedy seems to be a bit better than random, but no significant advantage is observed, which is due to the limited lookahead capabilities of the greedy heuristic. The results are visualized on the left in Fig. 7. The numbering of heuristics in figures is: "Heur 0" is the random heuristic, "Heur 1" is the greedy heuristic, and "Heur 2" is the BMC heuristic. The figure contains the cumulative number of detected mutants versus the amount of testing events. The mutants have been produced by varying the two last digits of the code and each mutant was ran 10 times with different random seeds.

Similar cases to this might of course occur in practice also. Consider, for example, a situation where a protocol contains complicated handshake before accessing the core functionality of the protocol. If random tester gets stuck with handshake, it will never test the actual functionality of the protocol. Also implementing the lookahead functionality with random walk algorithms instead of using bounded model checking might suffer similar problems.

We also analyzed how effective our approach is when testing conference protocol, which is more complicated that the simple combinatoric lock example. The results are on the right in Fig. 7. The mutants used are the same 25 **ioco** non-conforming mutants from [4, 6]. In the results we do not see dramatic advantage, although generally speaking the BMC and greedy heuristic are detecting the bugs faster, which is a promising result.

We provide per mutant results in Table 1. This table contains the average length of test sequence needed to detect a mutant (mutants have numeric names) using each of the

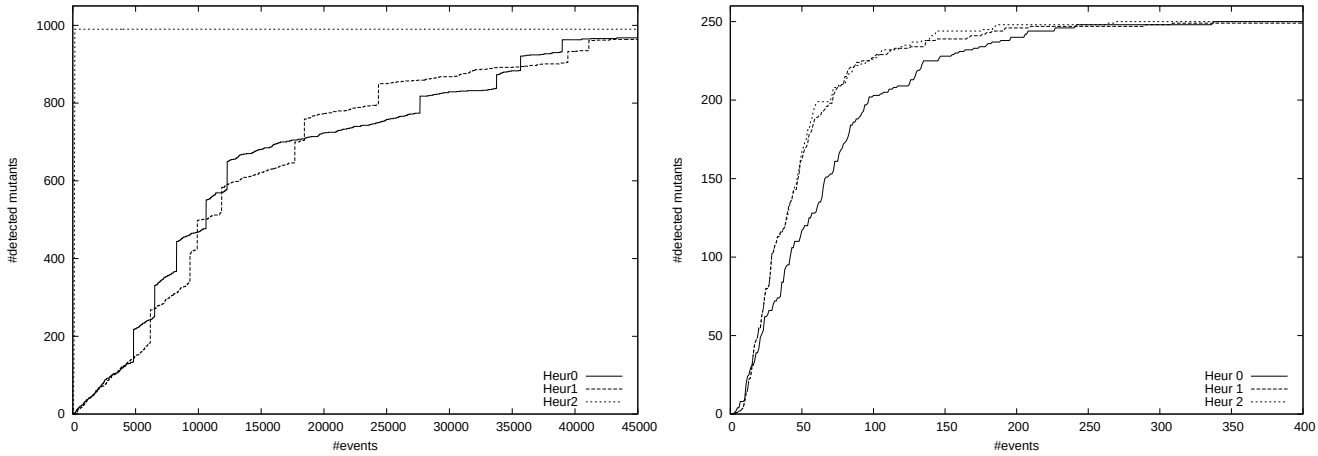

Figure 7. Cumulative number of detected mutants versus number of testing events. Combinatoric lock on the left, conference protocol on the right.

heuristics, and the ratio of test events required to detect a mutant on the average using BMC heuristic versus using a random heuristic. If we look at these results, we can see that the majority of mutants could in fact be detected faster on the average using a BMC based heuristic instead of a random one. However, there are mutants, which are actually detected slower than with random walk, for example mutant "398", which needs on the average roughly 10 percent more test events with BMC heuristic. Thus, indeed sometimes our coverage metric will work against us. Consider for example a cycle in the specification state space. If we traverse the cycle, our simple coverage metric will not consider it useful to to traverse it again before other parts of the specification have been covered. However, traversing the cycle might have left the internal state of the implementation corrupted and detectable only by traversing cycle again! Unfortunately, our coverage metric suggests us to explore other parts of the state space. Thus we draw the conclusion, that for the coverage based test selection to be truly useful, the used coverage metric should better match the expected failures to be detected. How this could be done is left for future work.

Table 1. Average number of required events to detect the mutants

Mutant	Heur 0	Heur 1	Heur 2	H2/H0 ratio
467	74.5	48.8	36.5	0.49
687	68.8	39.7	36.3	0.52
293	80.0	48.4	43.8	0.54
856	97.3	70.6	53.8	0.55
214	103.3	51.3	58.7	0.56
777	103.3	51.3	58.7	0.56
332	122.6	74.8	71.8	0.58
348	93.6	74.1	59.0	0.63
294	154.4	156.4	105.7	0.68
111	38.9	29.2	27.4	0.70
247	38.8	29.2	27.2	0.70
674	15.1	10.6	10.6	0.70
345	55.2	38.9	40.2	0.72
945	45.2	34.0	32.7	0.72
749	20.3	15.2	15.2	0.74
358	45.5	35.7	34.4	0.75
289	73.1	52.3	56.1	0.76
276	23.2	18.0	18.0	0.77
384	47.8	43.2	39.0	0.81
548	75.3	88.9	62.8	0.83
462	84.7	59.5	71.8	0.84
738	84.7	59.5	71.8	0.84
100	42.1	38.8	41.2	0.97
836	42.1	38.8	41.2	0.97
398	94.7	79.4	103.6	1.09

9 Conclusions

In this work a new test selection strategy for formal conformance testing is presented. The method is based on using specification coverage to guide test selection. The presented algorithm extends the on-the-fly testing algorithm of [4] with a specification coverage based test selection heuristic.

The test selection heuristic combines a greedy method with randomization to guarantee completeness. The greedy part of the implementation uses a bounded model checker

presented in [11] as a novel implementation technique.

To experiment with the proposed method we have developed an on-the-fly-test system. We were able to demonstrate cases where the suggested approach is very efficient. Furthermore, experiments on more real life case study, the conference protocol, demonstrate that the method is feasible, however, the achieved improvements are not as dramatic. We consider the initial results encouraging, while keeping in mind that the coverage metric employed was a very simple one.

As future work different specification coverage metrics should be considered, using some kind of hypothesis of the mutants to be detected by the method to select an appropriate coverage metric. Also a formal framework for comparing test selection strategies is required. We speculate that different formulations of two player games could be an interesting starting point for such a framework.

Acknowledgements

This work has been financially supported by Academy of Finland (Projects 47754, 53695) and Conformiq Software Ltd.

References

[1] B. Beizer. *Software testing techniques - 2nd ed.* International Thomson Computer Press, The United States of America, 2001.

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, pages 193–207. Springer, March 1999.

[3] R. Castanet and D. Rouillard. Generate certified test cases by combining theorem proving and reachability analysis. In *Testing of Communicating Systems XIV, Applications to Internet Technologies and Services, Proceedings of the IFIP 14th International Conference on Testing Communicating Systems*, pages 267–282, Berlin, Germany, March 2002.

[4] R. de Vries and J. Tretmans. On-the-fly conformance testing using SPIN. *Software Tools for Technology Transfer*, 2(4):382–393, March 2000.

[5] J. Desel and W. Reisig. Place/Transition Petri nets. In *Lectures on Petri Nets I: Basic Models*, number 1491 in Lecture Notes in Computer Science, pages 122–173. Springer, 1998.

[6] J. Feenstra. Conference protocol case study. WWW pages at address `http://www.cs.utwente.nl/ConfCase`, last updated 1999-11-18.

[7] L. Feijs, N. Goga, and S. Mauw. Probabilities in the torx test derivation algorithm. In *SAM2000 - 2nd Workshop on SDL and MSC*, pages 173–188, June 2000.

[8] L. M. G. Feijs, N. Goga, S. Mauw, and J. Tretmans. Test selection, trace distance and heuristics. In *Testing of Communicating Systems XIV, Applications to Internet Technologies and Services, Proceedings of the IFIP 14th International Conference on Testing Communicating Systems*, pages 267–282, Berlin, Germany, March 2002.

[9] A. Groce and W. Visser. Model checking Java programs using structural heuristics. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 12–21, July 2002.

[10] K. Heljanko. Bounded reachability checking with process semantics. In *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR'2001)*, volume 2154 of *Lecture Notes in Computer Science*, pages 218–232, Aalborg, Denmark, Aug. 2001. Springer.

[11] K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'2001)*, volume 2173 of *Lecture Notes in Artificial Intelligence*, pages 200–212, Vienna, Austria, Sept. 2001. Springer.

[12] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[13] M. Mäkelä. Maria: Modular reachability analyser for algebraic system nets. In *Proceedings of the 23rd International Conference on Application and Theory of Petri Nets (ICATPN'2002)*, number 2360 in Lecture Notes in Computer Science, pages 427–436, Adelaide, Australia, June 2002. Springer.

[14] A. Pretschner. Classical search strategies for test case generation with constraint logic programming. In *Proceedings of Formal Approaches to Testing of Software (FATES'01)*, pages 47–60, Aalborg, Denmark, August 2001.

[15] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Proceedings of the 13th IEEE Conference on Automated Software Engineering (ASE)*, 2001, October 1998.

[16] J. Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[17] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.

[18] G. Wimmel, H. Lötzbeyer, A. Pretschner, and O. Slotosch. Specification Based Test Sequence Generation with Propositional Logic. *Journal on Software Testing Verification and Reliability*, 10(4):229–248, 2000.