Specification-based Testing with Simulation Relations

Canh Minh Do, Kazuhiro Ogata
School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
Email: {canhdominh,ogata}@jaist.ac.jp

***Abstract*—We propose a concurrent program testing technique that is a specification-based one and uses simulation relations from concurrent programs to formal specifications. For a formal specification S, a concurrent program P and a simulation relation r from P to S, the proposed technique is outlined as follows: (1) state sequences $s_0, s_1, \ldots, s_n$ are generated from P, (2) state sequences $s''_0, s''_1, \ldots, s''_m$ for S are obtained by converting $s_0, s_1, \ldots, s_n$ with r and (3) it is checked that S can accept $s''_0, s''_1, \ldots, s''_m$. (1) is very crucial, but we first tackle (2) and (3) and then the present paper focuses on (2) and (3).**

***Keywords*-concurrent program testing; Maude; meta-programming; simulation-based testing; simulation relations**

I. Introduction

Major concepts of programming languages that can be used to write concurrent programs emerged in the 1980s and since nearly then studies on testing concurrent programs have been conducted. Arora, et al. have comprehensively surveyed testing concurrent programs [1]. They categorize it into eight classes: (a) reachability testing, (b) structural testing, (c) model-based testing, (d) mutation-based testing, (e) slicing-based testing, (f) formal method-based testing, (g) random testing, and (h) search-based testing. Model checking concurrent programs has been intensively studied, which may be classified into (c) and/or (f). Java Pathfinder (JPF) [2], [3] is such a model checker. Model checking is superior to the other testing techniques in that the former exhaustively checks all possible execution paths (or computations). However, model checking concurrent programs often encounters the notorious state explosion, which has not yet been conquered reasonably well.

We need a concurrent program testing technique that can scale reasonably well because most important software systems are in the form of concurrent programs, which are large-scale. We then propose a concurrent program testing technique in this paper toward this aim. The technique is a specification-based one. We suppose that programmers write concurrent programs based on formal specifications. The FeliCa team has demonstrated that use of formal specifications is useful as well as feasible in a practical setting [4]. Therefore, our assumption must be reasonable. Programmers need to comprehend formal specifications and must know their concurrent programs well and then they must be able to find some good relations between formal specifications and concurrent programs. Such relations should be simulation relations from the latter to the former. Then, we use such simulation relations to test concurrent programs.

Given a formal specification S, a concurrent program P and a simulation relation r from P to S, the proposed technique is outlined as follows: (1) state sequences $s_0, s_1, \ldots, s_n$ are generated from P, (2) state sequences $s''_0, s''_1, \ldots, s''_m$ for S are obtained by converting $s_0, s_1, \ldots, s_n$ with r and (3) it is checked that S can accept $s''_0, s''_1, \ldots, s''_m$. (1) is very crucial, but we first tackle (2) and (3) and then the present paper focuses on (2) and (3).

Our approach uses formal specifications to test concurrent programs. Testing programs based on formal specifications has been studied [5]. Among such techniques are a CSP-based one [6] and an Event-B model-based on [7]. One difference between existing such techniques and our approach is that test cases are generated from formal specifications in the former, while the counterparts are generated from concurrent programs.

The rest of the paper is organized as follows: § II Preliminaries, § III Toward Concurrent Program Testing, § IV Specification Testing with Simulation Relations, § V Experiments, and § VI Conclusion.

II. Preliminaries

A state machine $M \triangleq \langle S, I, T \rangle$ consists of a set S of states, the set $I \subseteq S$ of initial states and a binary relation $T \subseteq S \times S$ over states. $(s, s') \in T$ is called a state transition and may be written as $s \rightarrow_M s'$. Let $\rightarrow^*_M$ be the reflexive and transitive closure of $\rightarrow_M$. The set $R_M \subseteq S$ of reachable states w.r.t. M is inductively defined as follows: (1) for each $s \in I$, $s \in R$ and (2) if $s \in R$ and $(s, s') \in T$, then $s' \in R$. A state predicate p is called invariant w.r.t. M iff $p(s)$ holds for all $s \in R_M$. A finite sequence $s_0, \ldots, s_i, s_{i+1}, \ldots, s_n$ of states is called a finite semi-computation of M if $s_0 \in I$ and $s_i \rightarrow^*_M s_{i+1}$ for each $i = 0, \ldots, n-1$. If that is the case, it is called that M can accept $s_0, \ldots, s_i, s_{i+1}, \ldots, s_n$.

Given two state machines M_C and M_A, a relation r over R_C and R_A is called a simulation relation from M_C to

This work was partially supported by JSPS KAKENHI Grant Number JP26240008 & JP19H04082.
DOI reference number: 10.18293/SEKE2019-027

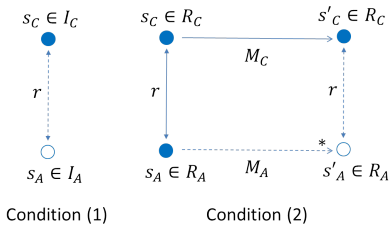

Figure 1. A simulation relation from M_C to M_A

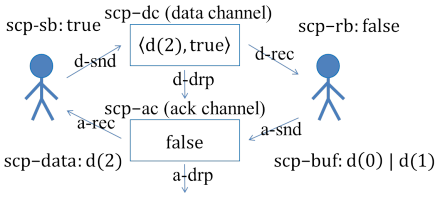

Figure 2. A state of SCP

M_A if r satisfies the following two conditions: (1) for each $s_C \in I_C$, there exists $s_A \in I_A$ such that $r(s_C, s_A)$ and (2) for each $s_C, s'_C \in R_C$ and $s_A \in R_A$ such that $r(s_C, s_A)$ and $s_C \rightarrow_{M_C} s'_C$, there exists $s'_A \in R_A$ such that $r(s_A, s'_A)$ and $s_A \rightarrow^*_{M_A} s'_A$ [8] (see Fig. 1). If that is the case, we may write that M_A simulates M_C with r. There is a theorem on simulation relations from M_C to M_A and invariants w.r.t M_C and M_A: for any state machines M_C and M_A such that there exists a simulation relation r from M_C to M_A, any state predicates p_C for M_C and p_A for M_A such that $p_A(s_A) \Rightarrow p_C(s_C)$ for any reachable states $s_A \in R_{M_A}$ and $s_C \in R_{M_C}$ with $r(s_C, s_A)$, if $p_A(s_A)$ holds for all $s_A \in R_{M_A}$, then $p_C(s_C)$ holds for all $s_C \in R_{M_C}$ [8]. The theorem makes it possible to verify that p_C is invariant w.r.t. M_C by proving that p_A is invariant w.r.t. M_A, M_A simulates M_C with r and $p_A(s_A)$ implies $p_C(s_C)$ for all $s_A \in R_{M_A}$ and $s_C \in R_{M_C}$ with $r(s_C, s_A)$.

States are expressed as braced soups of observable components, where soups are associative-commutative collections and observable components are name-value pairs in this paper. The state that consists of observable components oc_1, oc_2 and oc_3 is expressed as $\{oc_1\ oc_2\ oc_3\}$, which equals $\{oc_3\ oc_1\ oc_2\}$ and some others because of associativity and commutativity. We use Maude [9], a rewriting logic-based computer language, as a specification language because Maude makes it possible to use associative-commutative collections.

Simple Communication Protocol (SCP), a communication protocol, is used as one running example in this paper. SCP consists of a sender, a receiver and two channels between them. One channel called dc (data channel) is a cell that is used to transfer pairs $\langle d, b \rangle$, where d is a data value and b is a Boolean value, to the receiver from the sender, and the other channel called ac (ack channel) is a cell that is used to deliver Boolean values (as ack) to the sender from the receiver. Both cells are unreliable in that the contents may drop. The sender maintains two pieces of information that are sb (sender bit) and data. sb is a Boolean value and data is the data to be delivered next to the receiver. The receiver maintains two pieces of information that are rb (receiver bit) and buf. rb is Boolean value and buf is the list of data received so far. Initially, sb is true, data is d(0), rb is true, buf is empty, dc is empty and cc is empty. The sender has two actions to do that are d-snd and d-rec. d-snd does the following: the pair $\langle \text{data}, \text{sb} \rangle$ is put into dc. d-rec does the following: if ac has a Boolean value b, then b is extracted and if $b \neq$ sb, then data is set to the next data and sb is negated and otherwise nothing changes. The receiver has two actions to do that are a-snd and a-rec. a-snd does the following: rb is put into ac. a-rec does the following: if dc has $\langle d, b \rangle$, then $\langle d, b \rangle$ is extracted and if $b =$ rb, then d is added to buf at the end and rb is negated and otherwise nothing changes. There are two more actions that are d-drp and a-drp. d-drp does the following: if dc is not empty, dc becomes empty. a-drp does the following: if ac is not empty, ac becomes empty. Fig. 2 shows a state of SCP.

A state of SCP is expressed as follows:

```
{(scp-sb: b1) (scp-data: d(n)) (scp-rb: b2)
 (scp-buf: dl) (scp-dc: cell1) (scp-ac: cell2)}
```

Each of the six actions in SCP is formalized as state transitions, which are described in Maude (conditional) rewrite rules (or rules) as follows:

```
rl [d-snd] : {(scp-sb: B)(scp-data: D)
(scp-dc: DC) OCs}
=> {(scp-sb: B)(scp-data: D)
(scp-dc: (< D,B >)) OCs} .

crl [a-rec1] : {(scp-sb: B)(scp-data: d(N))
(scp-ac: B') OCs}
=> {(scp-sb: (not B))(scp-data: d(N + 1))
(scp-ac: empc) OCs} if B =/= B' .

crl [a-rec2] : {(scp-sb: B)(scp-data: D)
(scp-ac: B') OCs}
=> {(scp-sb: B)(scp-data: D)(scp-ac: empc)
OCs} if B = B' .

rl [a-snd] : {(scp-rb: B)(scp-ac: AC) OCs}
=> {(scp-rb: B)(scp-ac: B) OCs} .

crl [d-rec1] : {(scp-rb: B)(scp-buf: Ds)
(scp-dc: (< D,B' >)) OCs}
=> {(scp-rb: (not B))(scp-buf: (Ds | D))
(scp-dc: empc) OCs} if B = B' .

crl [d-rec2] : {(scp-rb: B)(scp-buf: Ds)
(scp-dc: (< D,B' >)) OCs}
=> {(scp-rb: B)(scp-buf: Ds)
(scp-dc: empc) OCs} if B =/= B' .
```

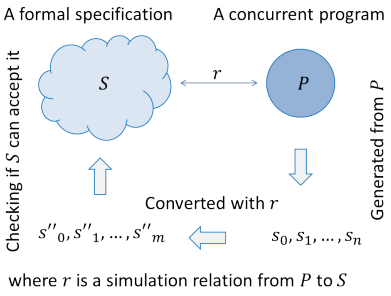

Figure 3. Specification-based concurrent program testing with a simulation relation

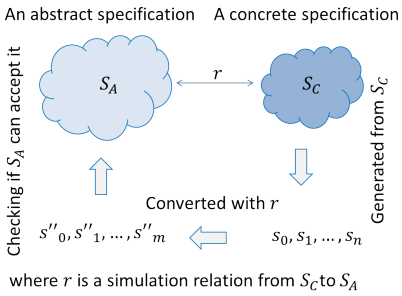

Figure 4. Specification-based specification testing with a simulation relation

```
rl [d-drp] : {(scp-dc: P) OCs}
=> {(scp-dc: empc) OCs} .

rl [a-drp] : {(scp-ac: B) OCs}
=> {(scp-ac: empc) OCs} .
```

The search command given by Maude can conduct reachability analysis of state machines specified in Maude. Let s_1 and s_2 be the following states:

```
{(scp-sb: true)(scp-data: d(0))
(scp-rb: false)(scp-buf: d(0))
(scp-dc: < d(0),true >)(scp-ac: false)}

{(scp-sb: false)(scp-data: d(1))
(scp-rb: false)(scp-buf: d(0))
(scp-dc: < d(0),true >)(scp-ac: false)}
```

The search command "`search [1,2] in SCP :` s_1 `=>*` s_2 `.`" checks if s_2 is reachable from s_1 in depth 2, where `SCP` is the module in which SCP is specified. If so, the command finds one transition sequence from s_1 to s_2. Because there is such a transition sequence, the commands finds it. If `[1,1]` is used instead of `[1,2]`, then because there is no transition sequence from s_1 to s_2 in depth 1, the command does not find any such transition sequences.

Maude has meta-programming (or reflexive programming) facilities with which we can develop software tools, such as Real-Time Maude. The first search command can be expressed in the term: `metaSearch(upModule('SCP, false), upTerm(`s_1`), upTerm(`s_2`), nil, '*, 2, 0)`, where `upModule` converts a module to its meta-representation, the term representing the module, and `upTerm` converts a term to its meta-representation. By reducing the term, we can essentially get the same result as the one obtained by the first search command.

III. Toward Concurrent Program Testing

Testing concurrent programs is inherently different from testing sequential programs. All we need to test the latter is basically to check the output for each input, although there is some room to generate better test cases. Even though there is no decidable test oracle, we could use the metamorphic testing technique. Concurrent programs have multiple active entities, such as threads, and therefore can lead to lots of execution scenarios. What execution scenario will be taken depends a scheduler, such as Java VM thread scheduler. It is usually impossible to control such a scheduler with an ordinal program. There may be some nondeterminism in concurrent programs. Hence, it does not suffice to have concurrent programs run on real machines to test such programs because we cannot check all possible execution scenarios.

One possible remedy is to use software model checkers, such as Java Pathfinder (JPF) [3]. JPF has its own VM running on a native Java VM and controls its own VM to cover all possible execution scenarios of concurrent programs. Many case studies with JPF have been reported and several techniques for JPF that may mitigate the notorious state explosion have been proposed. Even so, non-JPF experts can often encounter the state explosion that cannot be mitigated at all. We have written an ABP simulator in Java and tried to model check with JPF running on a computer that carries a 32GB memory the simulator in which each channel capacity is 3 and 3 messages are delivered to the receiver from the sender [10]. We have spent several days but the information we have obtained was out of memory. Thus, we do not think that it would suffice to use software model checkers, such as JPF, to test concurrent programs.

We propose a concurrent program testing technique that is a specification-based testing one (see Fig. 3). Let S be a formal specification of a state machine and P be a concurrent program. A state machine could be extracted from P, for example, as what is done by JPF. What we would like to do is to test if P is an implementation of S, or S simulates P. To this end, we use a simulation relation candidate r from P to S. For a formal specification S, a concurrent program P and a simulation relation r from P to S, the proposed technique does the following: (1) finite state sequences $s_1, s_2, \ldots, s_n$ are generate from P, (2) each s_i of P is converted to a state s'_i of S with r, (3) one of each two consecutive states s'_i and s'_{i+1} such that $s'_i = s'_{i+1}$ is deleted, (4) finite state sequences $s''_1, s''_2, \ldots, s''_m$ are then obtained

and (5) it is checked that $s''_1, s''_2, \ldots, s''_m$ can be accepted by S. We suppose that programmers write concurrent programs based on formal specifications, although it may be possible to generate concurrent programs (semi-)automatically from formal specifications in some cases. The FeliCa team has demonstrated that programmers can write programs based on formal specifications and moreover use of formal specifications can make programs high-quality. Therefore, our assumption is meaningful as well as feasible. If so, programmers must have profound enough understandings of both formal specifications and concurrent programs so that they can come up with simulation relation candidates from the latter to the former.

In our approach, state sequences generated from P are test cases. Therefore, it is really crucial that what state sequences are generated from P and how they are generated. The former and the latter have something to do with the quality of test cases and the scalability of our approach, respectively. Some may say that our approach has the same problems as software model checkers. Our approach never checks any properties while generating state sequences from P. It would take non-trivial time to check properties. Thus, we anticipate that our approach can scale better than software model checkers.

IV. Specification Testing with Simulation Relations

This paper mainly focuses on the left part of the diagram shown in Fig. 3. However, we need something, which is substituted for P, from which state sequences are generated. We use an abstract specification S_A as S and a concrete specification S_C as P in this paper (see Fig. 4).

It is often the case that any state sequences generated from S_C cannot be accepted by S_A. A simulation relation r from S_C to S_A is not necessarily a function from S_C states to S_A states or from S_A states to S_C states in general. Because states in S_C are often designed by refining those in S_A, however, we conjecture that r is a function from S_C states to S_A states in practice.

Given a finite sequence $s_0, s_1, \ldots, s_n$ of states generated from S_C, each state is converted into a state in S_A with r, generating $s'_0, s'_1, \ldots, s'_n$, where $s'_i = r(s_i)$ for each i and r is used as a function from S_C states to S_A states. There may be two consecutive states s'_i and s'_{i+1} such that $s'_i = s'_{i+1}$. If so, one of them is deleted. We then generate a sequence $s''_0, s''_1, \ldots, s''_m$ of states in S_A such that there does not exists i such that $s''_i = s''_{i+1}$ (see the bottom part of Fig. 4). We finally check if $s''_0, s''_1, \ldots, s''_m$ is a finite semi-computation of S_A (see the left part of Fig. 4).

Let `sim` be a function from S_C states to S_A. The function `simList` that converts $s_0, s_1, \ldots, s_n$ to $s''_0, s''_1, \ldots, s''_m$ is defined as follows:

```
eq simList(C | L, S)
= if S == {empty}
  then sim(C) | simList(L, sim(C))
  else (
   if compareTo(sim(C),S)
   then simList(L, S)
   else sim(C) | simList(L, sim(C)) fi ) fi .
```

where `_|_` is used as the constructor of state sequences. `compareTo(sim(C),S)` checks if there are two consecutive states such that they are equal.

Given a module `mQid` in which a state machine is specified, two states `S1` & `S2` and the depth `B`, the function `checkSttTrans` checks if `S2` is reachable from `S1` in `B` w.r.t. the state machine, which is defined as follows:

```
ceq checkSttTrans(mQid, S1, S2, B)
= if sttTrans? :: ResultTriple
  then true else false fi
if sttTrans? :=
   metaSearch(upModule(mQid, false),
    upTerm(S1), upTerm(S2), nil, '*, B, 0) .
```

`metaSearch` is used to check if `S2` is reachable from `S1` in `B` w.r.t. the state machine specified as `mQid`.

Given a module `mQid` in which a state machine is specified, a sequence of the state machine states and a depth `B`, the function `checkConform` checks if the state sequence is a finite semi-computation of the state machine, which is defined as follows:

```
eq checkConform(mQid, S1 | S2 | L, B)
= $checkConform(mQid, S2 | L, S1, 0, B).
eq $checkConform(mQid, nil, S, N, B)
= success .
eq $checkConform(mQid, S2 | L, S1, N, B)
= if checkSttTrans(mQid, S1, S2, B)
  then $checkConform(mQid, L, S2, N + 1, B)
  else {msg: "Failure",from: S1,to: S2,
        index: N, bound: B} fi .
```

`checkSttTrans(mQid, S1, S2, B)` checks if `S1` $\rightarrow^*_{\texttt{mQid}}$ `S1` in the depth `B`.

V. Experiments

We use a specification of Alternating Bit Protocol (ABP) as S_C. ABP is a communication protocol as SCP. A state in ABP is shown in Fig. 5. The difference between ABP and SCP is as follows: instead of the two cells used in SCP, ABP uses as two channels two queues that are unreliable in that

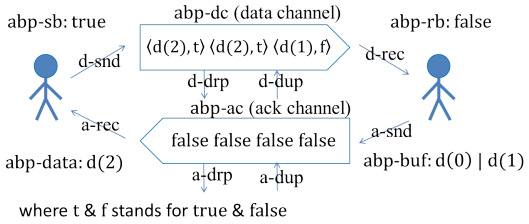

Figure 5. A state of ABP

an element in the queues may drop and/or be duplicated. Therefore, there are two actions for each queue: d-drp and d-dup for dc and a-drp and a-dup for ac. There are totally eight actions in ABP.

A state of ABP is expressed as follows:

```
{(abp-sb: b1) (abp-data: d(n)) (abp-rb: b2)
 (abp-buf: dl) (abp-dc: q1) (abp-ac: q2)}
```

Each of the eight actions in ABP is formalized as state transitions, which are described in Maude rules as follows:

```
rl [d-snd] : {(abp-sb: B)(abp-data: D)
(abp-dc: Ps) OCs}
=> {(abp-sb: B)(abp-data: D)
(abp-dc: (Ps | < D,B >)) OCs} .

crl [a-rec1] : {(abp-sb: B) (abp-data: d(N))
(abp-ac: (B' | Bs)) OCs}
=> {(abp-sb: (not B)) (abp-data: d(N + 1))
(abp-ac: Bs) OCs} if B =/= B' .

crl [a-rec2] : {(abp-sb: B) (abp-data: D)
(abp-ac: (B' | Bs)) OCs}
=> {(abp-sb: B) (abp-data: D)
(abp-ac: Bs) OCs} if B = B' .

rl [a-snd] : {(abp-rb: B)(abp-ac: Bs) OCs}
=> {(abp-rb: B) (abp-ac: (Bs | B)) OCs} .

crl [d-rec1] : {(abp-rb: B)(abp-buf: Ds)
(abp-dc: (< D,B' > | Ps)) OCs}
=> {(abp-rb: (not B))(abp-buf: (Ds | D))
(abp-dc: Ps) OCs} if B = B' .

crl [d-rec2] : {(abp-rb: B)(abp-buf: Ds)
(abp-dc: (< D,B' > | Ps)) OCs}
=> {(abp-rb: B)(abp-buf: Ds)(abp-dc: Ps)
OCs} if B =/= B' .

rl [d-drp] : {(abp-dc: (Ps1 | P | Ps2)) OCs}
=> {(abp-dc: (Ps1 | Ps2)) OCs} .

rl [d-dup] : {(abp-dc: (Ps1 | P | Ps2)) OCs}
=> {(abp-dc: (Ps1 | P | P | Ps2)) OCs} .

rl [a-drp] : {(abp-ac: (Bs1 | B | Bs2)) OCs}
=> {(abp-ac: (Bs1 | Bs2)) OCs} .

rl [a-dup] : {(abp-ac: (Bs1 | B | Bs2)) OCs}
=> {(abp-ac: (Bs1 | B | B | Bs2)) OCs} .
```

Let `sim` be a simulation function that converts an ABP state

```
(abp-sb: S)(abp-data: D)(abp-rb: R)
 (abp-buf: BUFF)(abp-dc: DC2)(abp-ac: AC2)
```

to a SCP state

```
{(scp-sb: S)(scp-data: D)(scp-rb: R)
(scp-buf: BUFF)(scp-dc: norm(hd(DC2)))
(scp-ac: norm(hd(AC2)))}
```

where `hd` returns the top element if a given queue is not empty and an error element otherwise and `norm` returns the cell in which the element is stored if the argument is an element and the empty cell if it is an error element.

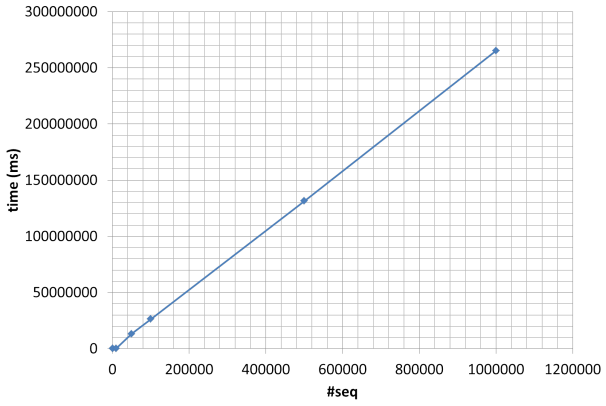

Figure 6. Time taken when the length of each state sequence is fixed (100) and the number of state sequences is changed (100, 1000, 10000, 50000, 100000, 500000 & 1000000)

Let `seqABP100` be a sequence of states randomly generated from the ABP specification such that its length is 100. Let `seqSCP` be the sequence of states obtained by `simList(seqABP100, {empty})`. We can check if `seqSCP` is a finite semi-computation of the SCP specification in the depth 2 by `checkConform('SCP, seqSCP, 2)`. The result is `success`. If we use 1 as the depth instead of 2, we get the following result:

```
{msg: "Failure",from: {scp-sb: true scp-data:
d(0) scp-rb: false scp-buf: d(0) scp-dc:
< d(0),true  > scp-ac: false},to: {scp-sb:
false scp-data: d(1) scp-rb: false scp-buf:
d(0) scp-dc: < d(0),true > scp-ac: false},
index: 20,bound: 1}
```

This is because two state transitions need to be taken to move the state following `from:` to the state following `to:` in SCP as described in Sect. II.

It is also important to know how many state transitions need to be taken to move one state to the next state in the state sequence of S_A obtained by converting a state sequence of S_C with a simulation function. We also conjecture that programmers who have written a concurrent program based on a formal specification can guess such information because they need to understand the formal specification well and know a simulation relation from the concurrent program to the formal specification.

We used the SCP and ABP specifications in Maude to measure time taken to generate state sequences from the ABP specification, transform them with the simulation relation from ABP to SCP to other state sequences, and check if the state sequences obtained can be accepted by the SCP specification. We used one node of SGI UV3000

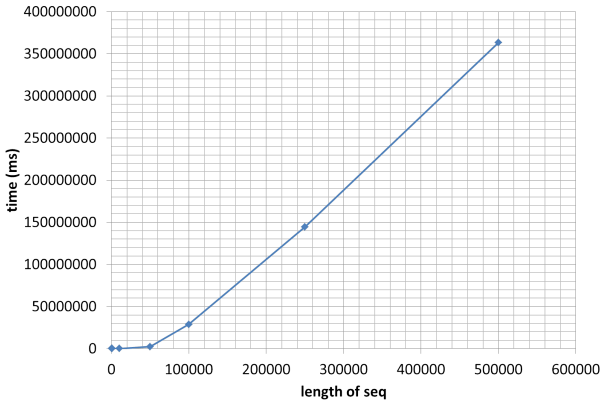

Figure 7. Time taken when the number of state sequences is fixed (1) and the length of the state sequence is changed (100, 1000, 10000, 50000, 100000, 250000 & 500000)

that carries 2.90GH microprocessor and 256GB memory for the experiments. Two sets of experiments were conducted. One set is to fix the length of each state sequence, which is 100, and modify the number of state sequences generated, which is one of 100, 1000, 10000, 50000, 100000, 500000 and 1000000. The other set is to fix the number of state sequences generated, which is one, and modify the length of the state sequence, which is one of 100, 1000, 10000, 50000, 100000, 250000 and 500000. For both sets of experiments, 2 was used as the depth of state transitions. Fig. 6 shows the experimental results for the first set. The time taken increases almost linearly as the number of state sequences generated increases. Fig. 7 shows the experimental results for the second set. The time taken increases a bit greater than linearly as the length of the state sequence generated increases.

VI. Conclusion

We have proposed a concurrent program testing technique that is a specification-based one and uses simulation relations from concurrent programs to formal specifications. For a formal specification S, a concurrent program P and a simulation relation from P to S, the proposed technique is outlined as follows: (1) state sequences $s_0, s_1, \ldots, s_n$ are generated from P, (2) state sequences $s''_0, s''_1, \ldots, s''_m$ for S are obtained by converting $s_0, s_1, \ldots, s_n$ with r and (3) it is checked that S can accept $s''_0, s''_1, \ldots, s''_m$. The present paper has focused on (2) and (3).

The first set of experiments (shown in Fig. 6) indicates that it would be feasible to (almost) exhaustively check if state sequences whose length is small and that are generated from a concurrent program can be accepted by a formal specification with a simulation relation (candidate) from the program to the specification. This must be useful because of the small world hypothesis [11], which means that most flaws of programs lurk in a shallow depth and could be found with such exhaustive testing in a shallow depth. The second set of experiments (shown in Fig. 7) indicates that it would not be feasible to exhaustively test state sequences whose length is large and that are generated from a concurrent program with a formal specification and a simulation relation (candidate) from the program to the specification. There may be flaws lurking in programs in a non-shallow depth [12]. Therefore, it is worth testing state sequences whose length is large. It seems feasible to do so selectively. What and how long state sequences are selected is one piece of our future work.

The present paper does not mention anything about how to generate state sequences from concurrent programs. We plan to use Java as a programming language to write concurrent programs and to use JPF to generate state sequences from concurrent programs.

References

[1] V. Arora, R. K. Bhatia, and M. Singh, "A systematic review of approaches for testing concurrent programs," *Concurrency Computat.: Pract. Exper.*, vol. 28, no. 5, pp. 1572–1611, 2016.

[2] K. Havelund and T. Pressburger, "Model checking JAVA programs using JAVA PathFinder," *STTT*, vol. 2, no. 4, pp. 366–381, 2000.

[3] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda, "Model checking programs," *Autom. Softw. Eng.*, vol. 10, no. 2, pp. 203–232, 2003.

[4] T. Kurita, M. Chiba, and Y. Nakatsugawa, "Application of a formal specification language in the development of the "Mobile FeliCa" IC chip firmware for embedding in mobile phone," in *FM 2008*, 2008, pp. 425–429.

[5] M. Gaudel, "Software testing based on formal specification," in *PSSE 2007*, 2007, pp. 215–242.

[6] A. Cavalcanti and M. Gaudel, "Testing for refinement in CSP," in *ICFEM 2007*, 2007, pp. 151–170.

[7] D. H. Vu, A. H. Truong, Y. Chiba, and T. Aoki, "Automated testing reactive systems from Event-B model," in *4th NAFOSTED Conf. Info. & Comp. Sci.*, 2017, pp. 207–212.

[8] K. Ogata and K. Futatsugi, "Simulation-based verification for invariant properties in the OTS/CafeOBJ method," in *Refine 2007*, 2007, pp. 127–154.

[9] M. Clavel, et al., *All About Maude*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4350.

[10] K. Ogata, "Model checking designs with CafeOBJ – a contrast with a software model checker," Workshop on Formal Method and Internet of Mobile Things, ECNU, Shanghai, China, 2014.

[11] D. Jackson, *Software Abstraction*. The MIT Press, 2012.

[12] K. Ogata, M. Nakano, W. Kong, and K. Futatsugi, "Induction-guided falsification," in *8th ICFEM*, 2006, pp. 114–131.