Received February 7, 2022, accepted February 25, 2022, date of publication February 28, 2022, date of current version March 10, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3155629

Parallel Specification-Based Testing for Concurrent Programs

CANH MINH DO AND KAZUHIRO OGATA
School of Information Science, Japan Advanced Institute of Science and Technology (JAIST), Nomi, Ishikawa 923-1292, Japan

Corresponding author: Canh Minh Do (canhdominh@jaist.ac.jp)

This work was supported in part by the Japan Society for the Promotion of Science (JSPS) KAKENHI under Grant JP19H04082.

ABSTRACT The paper proposes a new testing technique for concurrent programs. The technique is a specification-based testing. For a formal specification S and a concurrent program P, state sequences are generated from P and checked to be accepted by S. We suppose that S is specified in Maude and P is implemented in Java. Java Pathfinder (JPF) and Maude are then used to generate state sequences from P and to check if such state sequences are accepted by S, respectively. Even without checking any property violations with JPF, JPF often encounters the notorious state space explosion while only generating state sequences. Thus, we propose a technique to generate state sequences from P and check if such state sequences are accepted by S in a stratified way. A tool is developed to support the proposed technique that can be processed naturally in parallel. Some experiments demonstrate that the proposed technique mitigates the state space explosion, which cannot be achieved with the straightforward use of JPF.

INDEX TERMS Simulation, divide & conquer approach, parallel algorithms, concurrent programs, specification-based testing.

I. INTRODUCTION

Studies on testing concurrent programs [1] have been conducted for nearly 40 years or even more. Compared to testing techniques for sequential programs, however, any testing techniques for concurrent programs do not seem mature enough. Moreover, many important software systems, such as operating systems, are in the form of concurrent programs. Therefore, testing techniques for concurrent programs must be worth studying so that they can be matured enough.

For a formal specification S and a (concurrent) program P, to test P based on S, we can basically take each of the following two approaches: (1) P is tested with test cases generated from S and (2) it is checked that state sequences generated from P can be accepted by S. The two approaches would be complementary to each other. Approach (1) checks if P implements the functionalities specified in S, while approach (2) checks if P never implements what is not specified in S. In terms of simulation, approach (1) checks if P can simulate S, while approach (2) checks if S can simulate P. Approaches (1) and (2) are often used in the program testing community and the refinement-based formal methods community, respectively, while both (1) and (2), namely bi-simulation, are often used in process calculi. The present paper proposes a testing technique for concurrent programs based on approach (2) mainly because P is a concurrent program and then could produce many different executions due to the inherent nondeterminacy of P, which often leads to subtle bugs in the program. Furthermore, we would like to check whether or not the program P conforms to the specification S.

In correct-by-construction system or software development, a system is initially specified in a very abstract formal specification and then incrementally developed to a concrete formal specification through a sequence of refinement steps. In each refinement step, we add details about data structures and algorithms so that the final formal specification is closer to the implementation or program from which the program can be implemented by human programmers or generated by automatic code generators [2]–[4]. Among formal methods in which stepwise refinement plays a crucial role are Vienna Development Method (VDM) (or VDM++) [5], Z method [6], Abstract State Machine (ASM) [7], B method [8], and Event-B [9]. The final specification can be verified by verifying each individual refinement step, while the program generated needs to be verified by proving the very final

The associate editor coordinating the review of this manuscript and approving it for publication was Mahmoud Elish.

refinement step from the final formal specification to the program implemented or generated based on the final formal specification or the automatic code generator, respectively. It is very hard to verify the correctness of automatic code generators [10], and it is also hard to verify the very final refinement step partly because we need to have full precise formal semantics of programming languages. However, we need to have a reasonably good technique that can be used to check the program implemented or generated against the final formal specification. Our specification-based testing technique can be regarded as a complement to the very last step in correct-by-construction software development to guarantee the correctness of the program with the specification.

In the present paper, we specify a formal specification S in Maude [11] while a program P developed based on the formal specification is implemented in Java. Java Pathfinder (JPF) [12] is an extensible model checker for Java programs so that we can interact with JPF while model checking a program. Hence, we use JPF to generate state sequences from P. Maude is equipped with reflective programming (meta-programming) facilities, making it possible to check whether a state sequence can be accepted by a formal specification [13], and so we use Maude to check if such state sequences are accepted by S. Most model checkers suffer from the notorious state space explosion problem and JPF is no exception. Hence, JPF often encounters the notorious state space explosion while generating state sequences even without checking any property violations. Because there could be a huge number of different states reachable from an initial state that could make a huge number of different state sequences generated due to the inherent nondeterminacy of concurrent programs. In addition, a whole big heap mainly constitutes one state in a program under test by JPF. Thus, we propose a technique to generate state sequences from P and check if such state sequences are accepted by S in a stratified way. The reachable state space from each initial state is divided into multiple layers, generating multiple sub-state spaces. We generate sub-state sequences for each sub-state space instead of the original reachable state space. If each sub-state space is much smaller than the original reachable state space, then it is feasible to generate its sub-state sequences even though it is infeasible to generate state sequences for the original reachable state space due to the state space explosion problem. Besides, we do not need to combine such sub-state sequences to obtain the original state sequences for our technique, but it suffices to check each sub-state sequence separately.

Let us suppose that each layer l has depth $d(l)$. Let $d(0)$ be 0. For each layer l, state sequences $s^l_0, \ldots, s^l_{d(l)}$ whose depth is $d(l)$ are generated from each state at depth $d(0) + \ldots + d(l-1)$ from P. Each s^l_i is converted into the state representation $f(s^l_i)$ used in S, where f is a simulation relation candidate from P to S. We conjecture that if S is refined enough, f would be an identity function. There may be adjacent states $f(s^l_i)$ and $f(s^l_{i+1})$ such that $f(s^l_i)$ is the same as $f(s^l_{i+1})$. If so, one of them is deleted. We then have state sequences $f(s^l_0), \ldots, f(s^l_N)$, where the number $N+1$ of the states in the sequence is usually much smaller than $d(l) + 1$ because execution units in P are much finer than those in S. We check if each $f(s^l_0), \ldots, f(s^l_N)$ is accepted by S with Maude [13]. The proposed technique is called a divide & conquer approach to testing concurrent programs. Generating sub-state sequences from each sub-state space in one layer are basically independent. This is one of the most important advantages of the technique, making it possible to generate sub-state sequences and check them with Maude in parallel. We develop a tool supporting the proposed technique in Java that can be proceeded in parallel. Some experiments demonstrate that the proposed technique mitigates the state space explosion, which cannot be achieved with the straightforward use of JPF. In summary, the present paper makes the following contributions:

- A new testing technique for concurrent programs to check the correctness of the program with the specification that can be used to complement the very last step in correct-by-constructions software development. For programmers who prefer developing programs based on specifications from scratch, our technique is fruitful to be used to verify that programs conform to specifications.
- A divide & conquer approach to testing concurrent programs that can mitigate the state space explosion problem and be naturally parallelized to improve the running performance of model checking. A tool is developed to support the technique.
- Experimental results show that the approach can detect bugs and mitigate the state space explosion in model checking. Our tool and case studies are publicly available at the webpage.[1]

The present paper is an extended and improved version of our workshop paper [14] with the following improvements:

- We improve our tool so that it can check state sequences generated by JPF with Maude on the fly in parallel instead of storing such state sequences to disk and checking it with Maude subsequently.
- We describe how our specification-based technique can be regarded as a complement to the very last step in correct-by-construction software development to guarantee the correctness of the program with the specification.
- We demonstrate how programmers can know the maximum number of step transitions in our technique with a case study, namely Alternating Bit Protocol (ABP).
- Finally, we conduct more case studies to demonstrate the usefulness of our tool as well as our technique/approach.

The rest of the paper is organized as follows: § II Preliminaries, § III Specification-based Concurrent Program Testing with a Simulation Relation, § IV State Sequence Generation from Concurrent Programs, § V A Divide & Conquer

[1]https://github.com/canhminhdo/spec-based

Approach to Generating State Sequences, § VI A Divide & Conquer Approach to Testing Concurrent Programs, § VII Case Studies: Alternating Bit Protocol (ABP), CloudSync Protocol, NSLPK Protocol, § VIII Related Work and § IX 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 said 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 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 $r(s'_C, s'_A)$ and $s_A \rightarrow^*_{M_A} s'_A$ [15] (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 $s_C \in R_{M_C}$ [15]. 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 $r(s_C, s_A)$. In this paper, M_A is specified in Maude, while M_C is implemented in Java. The proposed testing technique checks if M_A simulates M_C with a simulation relation candidate r such that r is an identity function or almost an identity function such that invariants w.r.t. M_A are preserved by r. If the proposed technique detects that M_A does not simulate M_C, M_C is likely not to satisfy some desired invariants. Otherwise, M_C enjoys desired invariants in the reachable state space tested.

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 [11], a rewriting logic-based computer language, as a specification language because Maude makes it possible to use associative-commutative collections. State transitions are specified by rewrite rules in Maude.

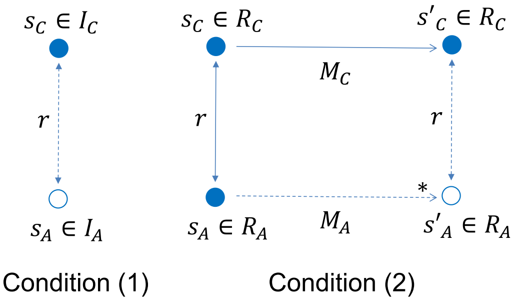

FIGURE 1. A simulation relation from M_C to M_A.

Let us consider an example a mutual exclusion protocol (the *test&set* protocol) in which the atomic instruction *test&set* is used. The protocol written in Algol-like pseudo-code is as follows:

```
Loop : "Remainder Section (RS)"
  rs : repeat while $test\&set(lock) = true$;
    "Critical Section (CS)"
  cs : $lock := false$;
```

lock is a Boolean variable shared by all processes (or threads) participating in the protocol. *test&set*(*lock*) does the following atomically: it sets *lock* to *true* and returns the old value stored in *lock*. Each process is located at either *rs* (remainder section) or *cs* (critical section). Initially, each process is located at *rs* and *lock* is *false*. When a process is located at *rs*, it does something (which is abstracted away in the pseudo-code) that never requires any shared resources; if it wants to use some shared resources that must be used in the critical section, then it performs the **repeat while** loop. It waits there while *test&set*(*lock*) returns *true*. When *test&set*(*lock*) returns *false*, the process is allowed to enter the critical section. The process then does something (which is also abstracted away in the pseudo-code) that requires to use some shared resources in the critical section. When the process finishes its task in the critical section, it leaves there, sets *lock* to *false* and goes back to the remainder section.

When there are three processes $p1$, $p2$, and $p3$, each state of the protocol is formalized as a term $\{(lock : b)\ (pc[p1] : l_1)\ (pc[p2] : l_2)\ (pc[p3] : l_3)\}$, where b is a Boolean value and each l_i is either *rs* or *cs* for i = 1,2,3. Initially, b is *false* and each l_i is *rs*. The state transitions are formalized as two rewrite rules. One rewrite rule says that if b is *false* and l_i is *rs*, then b becomes *true*, l_i becomes *cs*, and any other l_j (such that $j \neq i$) does not change. The other rewrite rule says that if l_i is *cs*, then b becomes *false*, l_i becomes *rs* and, any other l_j (such that $j \neq i$) does not change. The two rules are specified in Maude as follows:

```
rl [enter] : {(lock: false) (pc[I]: rs) OCs}
  => {(lock: true) (pc[I]: cs) OCs} .
rl [leave] : {(lock: B) (pc[I]: cs) OCs}
  => {(lock: false) (pc[I]: rs) OCs} .
```

where `enter` and `leave` are the labels (or names) given to the two rewrite rules, `I` is a Maude variable of process IDs, `B` is a Maude variable of Boolean values, and `OCs` is a Maude variable of observable component soups. `OCs` represents the remaining part (the other processes but process `I`) of the system. Both rules never change `OCs`. Let $S_{t\&s}$ refer to the specification of the *test&set* protocol in Maude.

III. SPECIFICATION-BASED CONCURRENT PROGRAM TESTING WITH A SIMULATION RELATION

We have proposed a new testing technique for concurrent programs that is a specification-based one and uses a simulation relation candidate from a concurrent program to a formal specification [13]. The technique is depicted in Fig. 2. Let S be a formal specification of a state machine and P be a concurrent program. Let us suppose that we know a simulation relation candidate r from P to S. The proposed technique does the following: (1) finite state sequences $s_1, s_2, \ldots, s_n$ are generated 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, where $s''_i \neq s''_{i+1}$ for each $i = 1, \ldots, m-1$, 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 [16]. 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. Even though consecutive equal states except for one are deleted, generating $s''_1, s''_2, \ldots, s''_m$ such that $s''_i \neq s''_{i+1}$ for each $i = 1, \ldots, m-1$, there may not be exactly one transition step but zero or more transition steps so that s''_i can reach s''_{i+1} w.r.t. P. Programmers are able to know the maximum number of atomic code fragments in programs for one atomic state transitions in formal specifications if programs are implemented/generated from formal specifications based on some reasonable techniques. EventB2Java [3] can generate concurrent Java programs from Event-B models in which actions of each event are translated to one atomic fragment and a shared lock is used to guarantee that at most one atomic fragment (or an event) can be executed at the same time by multiple threads in programs. Hence, one possible way to implement programs from formal specifications is to use a shared lock by multiple threads and translate each state transition in specifications to one atomic fragment in programs. If so, the maximum number of such transition steps in specifications is one. However, we can use more than one lock to implement programs. For ABP protocol implementation, we use two locks shared by multiple threads in the program as shown in Algorithm 2. One atomic fragment protected by one lock and another atomic fragment protected by the other lock can be executed in parallel. Hence, at most two atomic code fragments may be executed by multiple threads in parallel (or simultaneously), although each atomic action (or state transition) in the formal specification is implemented by one atomic code fragment in the program. In general, the number of locks used is the maximum steps in programs for one atomic state transition in formal specifications. We suppose that P is written in Java and Java Pathfinder (JPF) is used to generate state sequences from P [14].

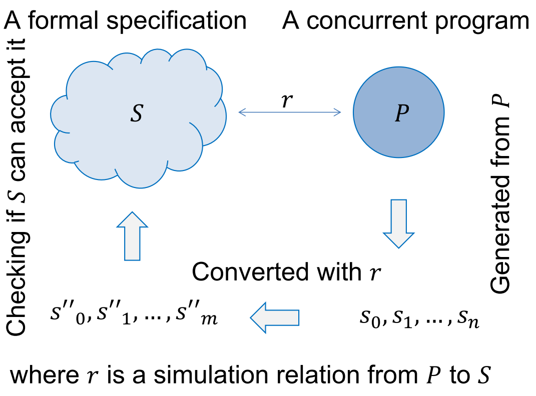

FIGURE 2. Specification-based concurrent program testing with a simulation relation.

IV. STATE SEQUENCE GENERATION FROM CONCURRENT PROGRAMS

A. JAVA PATHFINDER (JPF)

JPF is an extensible software model checking framework for Java bytecode programs that are generated by a standard Java compiler from programs written in Java. JPF has a special Virtual Machine (VM) in it to support model checking of concurrent Java programs, being able to detect some flaws lurking in concurrent Java programs, such as race conditions and deadlocks. When a flaw is detected, it reports a whole execution leading to the flaw. JPF explores all potential executions of a program under test, while an ordinary Java VM executes the code in only one possible way. JPF is basically able to identify points that represent execution choices in a program under test from which the execution could proceed differently.

Although JPF is a powerful model checker for concurrent Java programs, its straightforward use does not scale well and often encounters the notorious state space explosion. We anticipated previously [13] that we might mitigate the state space explosion if we do not check anything while JPF explores a program under test to generate state sequences. It is, however, revealed that we could not escape the state space explosion just without checking anything during the exploration conducted by JPF. This is because a whole big heap mainly constitutes one state in a program under test by JPF, while one state is typically expressed as a small term in formal specifications. The present paper then proposes a divide & conquer approach to generating state sequences from a concurrent program in a stratified way.

B. GENERATING STATE SEQUENCES BY JPF

JPF consists of two main components: (1) a VM and (2) a search component. The VM is a state generator. It generates state representations by interpreting Java bytecode instructions. A state is mainly constituted of a heap and threads plus an execution history (or path) that leads to the state. Each state is given a unique ID number. The VM implements a state management that makes it possible to do state matching, state storing, and execution backtracking when exploring a state space. Three key methods of the VM are employed by the search component:

- **forward** - it generates the next state and reports if the generated state has a successor; if so, it stores the successor on a backtrack stack for efficient restoration;
- **backtrack** - it restores the last state on the backtrack stack;
- **restoreState** - it restores an arbitrary state.

At any state, the search component is responsible for selecting the next state on which the VM should work, either by directing the VM to generate the next state (forward) or by telling it to backtrack to a previously generated one (backtrack). The search component works as a driver for the VM. There are some strategies that can be used to traverse the state space. By default, the search component uses depth-first search (DFS), although we can configure to use different strategies, such as breadth-first search.

The most important extension mechanism of JPF is listeners which provide a way to observe, interact with, and extend JPF execution. We can configure JPF with many of our own listener classes provided that our own listener classes need to extend the ListenerAdapter class. The ListenerAdapter class consists of all event notifications from the VMListener and SearchListener classes. It allows us to subscribe to VMListener and SearchListener event notifications by overriding some methods, such as:

- **searchStarted** - it is invoked when JPF has just entered the search loop but before the first forward;
- **stateAdvanced** - it is invoked when JPF has just got the next state;
- **stateBacktracked** - it is invoked when JPF has just backtracked one step;
- **searchFinished** - it is invoked when JPF is just done.

A *SequenceState* class that extends *ListenerAdapter* class is made to observe and interact with JPF execution. In *SequenceState* class, we override the two important methods: *stateAdvanced* and *stateBacktracked*. Whenever the *stateAdvanced* method is invoked, we need to retrieve all necessary information about the next state at this step. We use an instance *Path* of *ArrayList* class to maintain the path from the beginning state to the current state being visited by the DFS. Each element of *Path* corresponds to a state in JPF and is encapsulated as an instance of a *Configuration* class prepared by us. Each element of *Path* only stores the information for our testing purpose, which is mainly the values of observable components. For example, the information for the *test&set* mutual exclusion protocol is as follows:

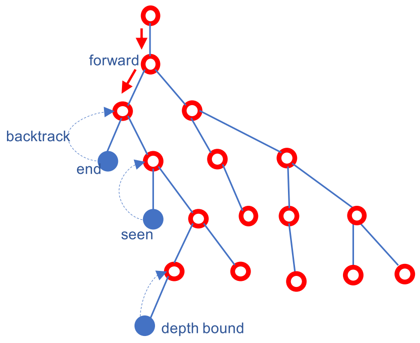

FIGURE 3. A way to generate state sequences with JPF.

- **stateId** - the unique id of a state;
- **depth** - the current depth of search path;
- **lock** - a *Lock* object that contains the *lock* observable component value, which is either *true* or *false*;
- **threads** - an *ArrayList* object of threads, each of which consists of the current location information that is either *rs* or *cs*.

We need to keep up with the change of observable components in each state stored in *Path*. Observable components are implemented as object data in JPF. To obtain that information, we need to look inside the heap of JPF. The heap contains a dynamic array of *ElementInfo* objects where the array indices are used as object reference values. An *ElementInfo* object contains a *Fields* object that actually stores the values of observable components. Thereby, we can gather the values of observable components, create a new *Configuration* object, and append it to *Path* whenever the *stateAdvanced* method is invoked.

Whenever JPF hits an end state, a state that has been already visited or a depth bound, a path (or a state sequence) is made, all but one are deleted from each of consecutive same states in the path. A cache is used to store all paths that have been explored. If the path does not exist in the cache, we save the path to the cache and then send it to another part of our system to check whether the path conforms to the specification with Maude on the fly subsequently. Otherwise, we just discard the path.

Because the reachable state space could be huge, we manage a parameter in order to prevent JPF from diverging as follows:

- **DEPTH** - the maximum depth from the initial state; once JPF reaches any state whose depth from the initial state is *DEPTH*, a backtrack message is sent to the search component for backtracking.

DEPTH could be set to unbounded, meaning that we ask JPF to generate as deep state sequences as possible. Every time JPF performs backtracking because of no more successor state, the last state is deleted from *Path* in the *stateBacktracked* method to keep up with the change of the current path in the DFS.

The way to generate state sequences from concurrent programs with JPF is depicted in Fig. 3. White nodes with a

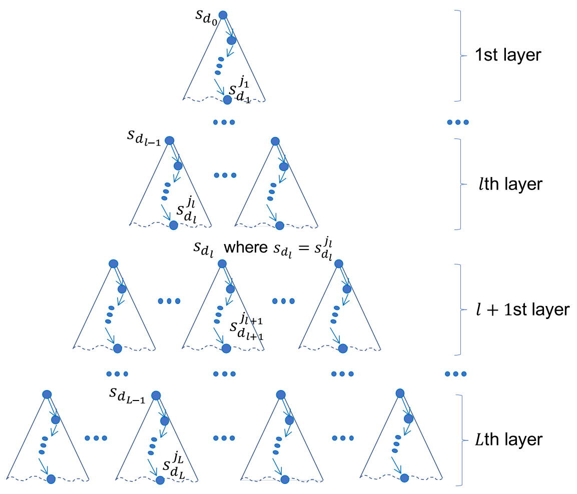

FIGURE 4. A divide & conquer approach to generating state sequences.

Algorithm 1: A Divide & Conquer Approach to Generating State Sequences for L Layers

```
input : P – a concurrent program Π_0 – the set of initial
        states of P d(1) ... d(L) – a list of non-zero
        natural numbers, where L is a non-zero natural
        number
output: a set of state sequences
1 forall $l \in 1 \ldots L$ do
2     $\Pi_l \leftarrow \emptyset$;
3     forall $\pi \in \Pi_{l-1}$ do
4         $Seq \leftarrow gen(last(\pi), d(l))$;
5         forall $\pi' \in Seq$ do
6             $\Pi_l \leftarrow \Pi_l \cup combine(\pi, \pi')$;
7 return $\Pi_L$;
```

thick border in red indicate that those nodes have been visited by JPF. Blue nodes cause backtracking because the node (or state) does not have any more successor node and such a node has been seen (or visited) before or the depth of the node reaches *DEPTH*.

V. A DIVIDE & CONQUER APPROACH TO GENERATING STATE SEQUENCES

JPF often encounters the notorious state space explosion even without checking any property violation while exploring a state space. When we do not set *DEPTH* to a moderately small number and ask JPF to exhaustively (or almost exhaustively) explore all (or a huge number of) possible states, JPF may not finish the exploration and may lead to out of memory. To mitigate the situation, the present paper proposes a technique to generate state sequences from concurrent programs in a stratified way, which is called a divide & conquer approach to generating state sequences. Given a concurrent program P, our approach splits the reachable state space from each initial state s_{d_0} into multiple layers, for example L layers (where L is a non-zero natural number) as shown in Fig. 4. Let $d(i)$ be the depth of layer i for $i = 0, 1, \ldots, L$. We suppose that there virtually exists layer 0 such that $d(0) = 0$. $d(i)$ is a non-zero natural number if $i = 1, \ldots, L$. Let d_i be $d(0) + \ldots + d(i)$ for $i = 0, \ldots, L$, namely that d_i is the depth of the bottom of layer i (or the depth of the top of layer $i + 1$) from the initial state. States located at the depth d_i are called beginning states of layer $i + 1$ (or ending states of layer i). The depth of a state is the depth from the initial state where the state is located. In Fig. 4, s_{d_i} denotes a beginning state of layer $i + 1$ for $i = 0, \ldots, L$, and $s^{j_i}_{d_i}$ denotes an ending state of layer i for $i = 0, \ldots, L$, although we do not use $s^{j_0}_{d_0}$ (which is an ending state of layer 0, a beginning state of layer 1, and the same as s_{d_0}) in the figure. $s^{j_i}_{d_i}$ is also a beginning state of layer $i + 1$, such as s_{d_l} that equals $s^{j_l}_{d_l}$ in Fig. 4.

Intuitively, we first generate state sequences from each initial state, where the length of each sequence is $d(1)$ (see Fig. 4). If $d(1)$ is small enough, it is possible to do so. We then generate state sequences from each of the states at depth $d(1)$, where the length of each sequence is $d(2)$. If $d(2)$ is small enough, it is also possible to do so. Given one initial state, there is one sub-state space in the first layer explored by JPF, while there are as many sub-state spaces in the second layer as the number of states at depth $d(1)$ reachable from the initial state. Combining each state sequence seq_1 in layer 1 and each state sequence seq_2 in layer 2 such that the last state of seq_1 equals the first state of seq_2 and either the last state of seq_1 or the first state of seq_2 is removed, we are to generate state sequences, where the length of each sequence is $d(1) + d(2)$, which can be done even though $d(1) + d(2)$ is large. Similarly, we could generate state sequences up to depth $d(1) + \ldots + d(L)$ for L layers (see Fig. 4).

Let Π_i for $i = 0, \ldots, L$ be the set of all state sequences generated in layer i reachable from initial states. Initially, Π_0 is the set of initial states whose depth is 0. For a state sequence $\pi \in \Pi_i$ for $i = 0, \ldots, L$, let $last(\pi)$ be the last state in π. For a state s_{d_i} at the bottom of the ith layer or the beginning of the $i+1$st layer (see in Fig. 4), let $gen(s_{d_i}, d(i+1))$ for $i = 0, \ldots, L - 1$ be the set of state sequences in the $i + 1$st layer reachable from s_{d_i}, where the length of each state sequence is $d(i + 1)$. For two state sequences π and π' such that the last state of π is equal to the first state of π', let $combine(\pi, \pi')$ be the combined state sequence of π and π', where either the last state of π or the first state of π' is removed. Algorithm 1 shows a divide & conquer approach to generating state sequences for L layers reachable from the set of initial states Π_0. For each layer $l \in 1 \ldots L$, Π_l is initially set to empty at line 2. Suppose that we have the set of state sequences Π_{l-1} of the $l - 1$st layer, which is obvious for layer 1 because Π_0 has initialized. For each state sequence π in the preceding layer, whose length is d_{l-1}, we get the last state of π to generate a set of state sequences reachable from $last(\pi)$, where the length of each state sequence is $d(l)$, and assign to *Seq* at line 4. If $d(l)$ is small enough, it is possible to do so. For each state sequence π' in *Seq*, we can combine the state sequence π and π' to get a state sequence of the lth layer, whose length is d_l, because the last state of π is equal to the first state of π', and add it to Π_l at line 6. Generating state sequences at each layer and combining with state sequences at the preceding layer, we are able to generate state sequences for L layers even though d_L is large. Π_L is returned as the final result of Algorithm 1 at line 7.

When generating state sequences for L or unbounded layers, *DEPTH* can be regarded as $d(1) + \ldots + d(L)$ or unbounded, respectively. When the entire reachable state space is huge, *DEPTH* parameter is also shared by many bounding techniques, which systematically explore a part of the entire reachable state space, such as bounded model checking (BMC) [17]–[19] and context bounding [20], [21]. In those existing studies, the depth parameter is iteratively increased until a bug or counterexample is found. In our environment, selecting *DEPTH*, or in other words, the depth of each layer is essential. We can select each layer depth as follows. We start with a small depth, namely ten, as the layer depth and increment it by one small number, namely five, until JPF is not able to explore the entire sub-state space up to the depth reachable from an initial state in a reasonable amount of time. The depth in which JPF is able to do so at the last should be used as each layer depth. In addition, the number of states located at the first layer for each depth option is also considered. However, finding good depth information for layers is one piece of our future work.

Let us consider the *test&set* protocol and suppose that we write a concurrent program (denoted $P_{t\&s}$) in Java based on the specification $S_{t\&s}$ of the protocol. We suppose that there are three processes participating in the protocol. $S_{t\&s}$ has one initial state and so does $P_{t\&s}$. Let each of $d(1)$ and $d(2)$ be 50 and let us use the proposed technique to generate state sequences from $P_{t\&s}$. One of the state sequences (denoted seq_1) generated in layer 1 is as follows:

```
{(pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs)
 (lock: false)} |
{(pc[p1]: rs) (pc[p2]: cs) (pc[p3]: rs)
 (lock: true)} |
{(pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs)
 (lock: false)} | nil
```

where _|_ is the constructor for non-empty state sequences and `nil` denotes the empty state sequence. Note that atomic execution units used in $P_{t\&s}$ are totally different from those used in $S_{t\&s}$. Therefore, the depth of layer 1 is 50, but the length of the state sequence generated is three. One of the state sequences (denoted seq_2) generated in layer 2 is as follows:

```
{(pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs)
 (lock: false)} |
{(pc[p1]: cs) (pc[p2]: rs) (pc[p3]: rs)
 (lock: true)} |
{(pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs)
 (lock: false)} | nil
```

Note that the last state in the first state sequence is the same as the first state in the second state sequence. Combining the two state sequences such that consecutive equal states are removed to withhold one, we get the combined state sequence (denoted seq_3) as follows:

```
{(pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs)
 (lock: false)} |
{(pc[p1]: rs) (pc[p2]: cs) (pc[p3]: rs)
 (lock: true)} |
{(pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs)
 (lock: false)} |
{(pc[p1]: cs) (pc[p2]: rs) (pc[p3]: rs)
 (lock: true)} |
{(pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs)
 (lock: false)} | nil
```

This is one state sequence generated from $P_{t\&s}$, where *DEPTH* is 100.

If each sub-state space is much smaller than the original reachable state space, then it is feasible to generate its sub-state sequences even though it is infeasible to generate state sequences for the original reachable state space due to the state space explosion problem. Therefore, using the divide & conquer approach to generating state sequences, we are able to generate longer or deeper state sequences that are unfeasible by using JPF only without our approach. In addition, generating state sequences for each sub-state space is independent from that for any other sub-state spaces. Especially for sub-state spaces in one layer, generating state sequences for each sub-state space is totally independent from that for each other. This characteristic of the proposed technique makes it possible to generate state sequences from concurrent programs in parallel. For example, once we have generated state sequences in layer l, we can generate state sequences for all sub-state spaces in layer $l + 1$ simultaneously. This is an advantage of the divide & conquer approach to generating state sequences from concurrent programs.

VI. A DIVIDE & CONQUER APPROACH TO TESTING CONCURRENT PROGRAMS AND THE SUPPORT TOOL

Once state sequences are generated from a concurrent program P, we check if a formal specification S can accept the state sequences with Maude on the fly and show the result. For example, we can check if seq_3 can be accepted by $S_{t\&s}$ with Maude. Instead of checking if seq_3 can be accepted by $S_{t\&s}$, however, it suffices to check if each of seq_1 and seq_2 can be accepted by $S_{t\&s}$.

For each layer l, we generate state sequences that start from each state located at depth $d(1) + \ldots + d(l - 1)$ from a concurrent program P with JPF and check if each state sequence generated in layer l can be accepted by a formal specification S with Maude. We could first generate all (sub-)state sequences from P in the stratified way and then could check if each state sequence can be accepted by S as shown in Algorithm 1. But, we do not combine multiple (sub-)state sequences to generate a whole state sequence of P because we do not need to do so and it suffices to check if each (sub-)state sequence can be accepted by S in order to check if a whole state sequence can be accepted by S. This way to generate (sub-)state sequences from P and to check if each

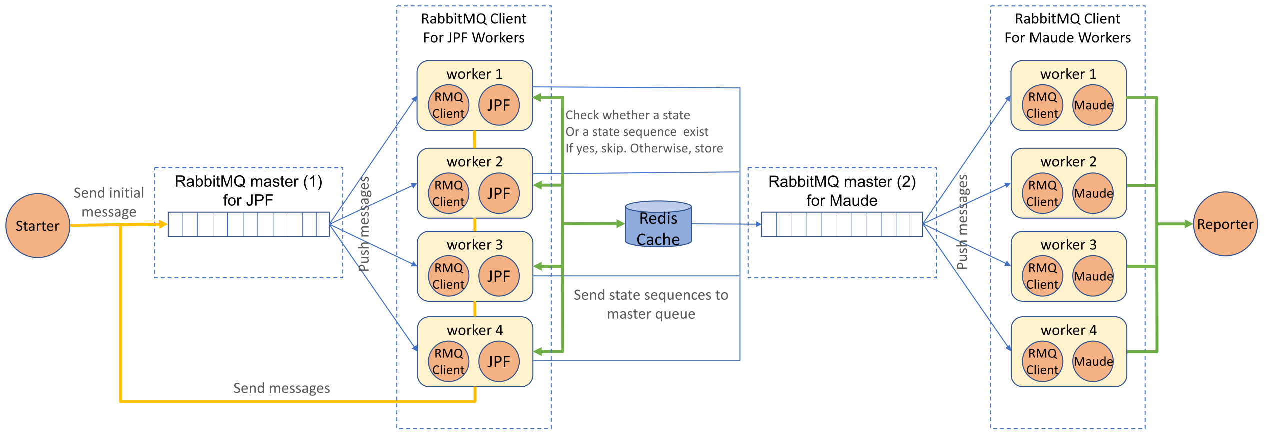
Starter
Send initial message
RabbitMQ master (1) for JPF
Push messages
RabbitMQ Client For JPF Workers
worker 1
worker 2
worker 3
worker 4
RMQ Client
JPF
Check whether a state Or a state sequence exist If yes, skip. Otherwise, store
Redis Cache
Send state sequences to master queue
Send messages
RabbitMQ master (2) for Maude
RabbitMQ Client For Maude Workers
Maude
Reporter

FIGURE 5. **The architecture of a tool supporting the proposed technique.**

(sub-)state sequence is accepted by S is called a divide & conquer approach to testing concurrent programs.

Our tool that supports the divide & conquer approach to testing concurrent programs has been implemented in Java. The tool architecture is depicted in Fig. 5 that is a master-worker model (or pattern), where there are two main groups. The first left half group uses one RabbitMQ master (1) and four JPF workers to generate state sequences while the second right half group also uses one RabbitMQ master (2) and four Maude workers to check whether or not state sequences can be accepted by specifications on the fly. Note that we can use as many workers as possible in each group. Both state sequence generation and conformance checking to specifications are conducted in parallel. We use Redis [22] and RabbitMQ [23] to develop our tool.

- Redis is an advanced key-value store and supports many different kinds of data structures, such as strings, lists, maps, sets, and sorted sets. It could hold its database entirely in memory as a big hash table. Redis is used as an efficient cache to avoid duplicating states and state sequences while generating state sequences.
- RabbitMQ is used as a message broker. The RabbitMQ master maintains a message queue to store and dispatch messages from/to RabbitMQ (RMQ) clients. In the first group, each JPF worker consists of a RabbitMQ client to fetch messages (states) from the RabbitMQ master (1). For a fetched message (a state), a JPF instance that is internally started by the worker starts generating state sequences from the state up to a depth. For each state sequence generated, we obtain the last state in the state sequence and check if the state exists in a cache of states. If not, the state is sent back to the RabbitMQ master (1). We jointly check if the state sequence exists in a cache of state sequences. If not, the state sequence is sent to the RabbitMQ master (2). In the second group, each Maude worker consists of a RabbitMQ client to fetch messages (state sequences) from the RabbitMQ master (2). For a fetched message (a state sequence), a Maude instance that is internally started by the worker checks if the state sequence can be accepted by a formal specification.

Initially, we run a starter program to send an initial message to the RabbitMQ master (1) for JPF to kick off the tool, where an initial message is regarded as an initial state specified in a specification. The starter program is just specialized in sending an initial message. First of all, we flush all keys and values from the Redis cache to clean up data in memory. Secondly, we make a connection to RabbitMQ master for JPF with a designated configuration. After making sure that it is connected, we prepare an initial message to send to the message queue maintained by RabbitMQ master (1). The data is encapsulated into a *Configuration* object, namely *config*. Before sending the initial message to the RabbitMQ master, we use *SerializationUtils* class supported by the Apache Commons Lang [24] to serialize the *config* object that makes it easy to deserialize to the original object at the receiver side without doing any extra thing.

As soon as the RabbitMQ master has received a message from a worker, the message is stored in a message queue. By default, the RabbitMQ master will pop a message from the message queue and then dispatch it to a worker, in sequence. RabbitMQ has a noticeable parameter that needs to be configured, namely, *prefetch*. It indicates a maximum of unacknowledged messages that each worker may receive at once. If *prefetch* is not configured, the default value is unlimited. From our experience, it takes much more time for JPF workers to generate state sequences than for Maude workers to check if state sequences are accepted by formal specifications. Hence, to improve the stability and efficiency of our environment, *prefetch* value is assigned to 1 and 10 for JPF workers and Maude workers, respectively.

Let us consider the first left half group in the environment architecture. Firstly, JPF workers make a connection to the RabbitMQ master (1). After connected, workers are willing to receive messages from RabbitMQ master (1). Whenever a worker receives a message from RabbitMQ master (1), the worker deserializes the message into its original object, namely *config*, which is an object of *Configuration* class, by invoking the *deserialize* method of *SerializationUtils* class. Given the *config* object, we create an instance of *RunJPF* class. Then the instance invokes the *run* method to initialize a JPF instance with some configuration that is built from the *config* object. We need to let the JPF instance know which message arguments are passed to the system under test and also need to register our listener class to the JPF instance so that we can interact with the JPF instance. Consequently, the worker can internally start a JPF instance to generate state sequences from the given message.

Note that all workers, as well as JPF instances, are running in parallel and using one shared Redis instance. A JPF instance traverses the (sub-)state space reachable from the state derived from a given message. Whenever a JPF instance reaches the designated depth or finds that the current state being visited has no more successor states, our listener class does the following:

- Removing all consecutive same states except for one from the state sequence;
- Converting the state sequence to a string representation, then using the SHA256 algorithm to hash the string representation to a unique signature;
- Asking the Redis cache whether the state sequence exists or not; If yes, skipping what follows; Otherwise, saving the signature as the key and the string representation as the value into the Redis cache, making a connection to the RabbitMQ master for Maude (2), and then sending the state sequence as a message to the RabbitMQ master (2) for conformance checking to formal specifications in the second group subsequently;
- Obtaining the last state from the state sequence, converting it to a string representation, and using the SHA256 algorithm to hash the string to a unique signature;
- Asking the Redis cache whether the state exists or not; If yes, skipping what follows; Otherwise, asking the Redis cache to save the signature as the key and the string representation for the last state as the value into the Redis cache and sending a message that contains the last state's information to RabbitMQ master (1), which then prepares a message that asks a worker to generate state sequences from the state unless the current layer is the final one.

Let us consider the second right half group where the tool has been integrated with Maude so that a Maude instance can check if state sequences are accepted by formal specifications on the fly. The RabbitMQ master (2) is used to gather all state sequences emitted from the workers in the first group. We have known that once JPF instances have generated state sequences, they send the state sequences to the RabbitMQ master (2) where a message queue is maintained to store such state sequences. The RabbitMQ master (2) then gradually dispatches state sequences as messages to Maude workers. Initially, each Maude worker makes a connection to the RabbitMQ master (2) with a designated configuration. After connected, the worker launches internally a Maude instance and feeds to the Maude instance some Maude files that are a specification of a concurrent program being tackled and a meta-programming script to check the correctness of state sequences. Whenever the worker receives a message, the worker deserializes the message into the original object that represents a state sequence. Then it calls to the Maude instance to check whether or not the state sequence is accepted by the specification loaded into the Maude instance before. Given a command line with a designated module name, a state sequence, and a depth, an instruction that can be fed into the Maude instance is constructed. Let `M` be a module name, `Seq` is a state sequence being checked, and `D` is a depth that is the possible number of transition steps (see § III). The instruction looks like as follows:

```
reduce checkConform(M, Seq, D) .
```

Whenever the Maude instance receives the instruction, the Maude instance executes it and checks whether `Seq` is accepted by `M` with depth `D`. A result will be returned in the form of either a *success* or *failure* message. As the worker that calls the Maude instance to check the state sequence receives the message result from the Maude instance, it parses the message to know what it is, and then displays the result to the console output. All state sequences and their results can be stored into MySQL [25] to easily monitor and diagnose problems if needed. Note that all workers are running in parallel and use different Maude instances but load the same specification and the meta-programming script.

VII. CASE STUDIES

We conducted case studies in which Alternating Bit Protocol (ABP), a cloud synchronization protocol (CloudSync) and Needham-Schroeder-Lowe Public-Key authentication protocol (NSLPK) were tackled. We experimented on two versions of CloudSync protocol, including Revised CloudSync and Original CloudSync that are described in detail in this section. Hence, we conducted four case studies in total. Our experiments were carried out by an Apple iMac Late 2015 that had Processor 4GHz Intel Core i7 and Memory 32GB 1867 MHz DDR3.

A. ALTERNATING BIT PROTOCOL (ABP)

1) INTRODUCTION

Alternating Bit Protocol (ABP) is a communication protocol and can be regarded as a simplified version of TCP. ABP makes it possible to reliably deliver data from a sender to a receiver even though two channels between the sender and receiver are unreliable in that elements in the channels may be dropped and/or duplicated. The sender maintains two pieces of information: *sb* that stores a Boolean value and *data* that stores the data to be delivered next. The receiver maintains two pieces of information: *rb* that stores a Boolean value and *buf* that stores the data received. One channel *dc* from the sender to the receiver carries pairs of data and Boolean values, while the other one *ac* from the receiver to the sender carries

Boolean values. There are two actions done by the sender: (sa1) the sender puts the pair (*data*, *sb*) into *dc* and (sa2) if *ac* is not empty, the sender extracts the top Boolean value *b* from *ac* and compares *b* with *sb*; if $b \neq sb$, *data* becomes the next data and *sb* is complemented; otherwise nothing changes. Actions (sa1) and (sa2) done by the sender are denoted *d-snd* and *a-rec*, respectively. There are two actions done by the receiver: (ra1) the receiver puts *rb* into *ac* and (ra2) if *dc* is not empty, the sender extracts the top pair (d, b) from *dc* and compares *b* with *rb*; if $b = sb$, *d* is stored in *buf* and *rb* is complemented; otherwise nothing changes. Actions (ra1) and (ra2) done by the receiver are denoted *a-snd* and *d-rec*, respectively. There are four more actions to *dc* and *ac* because the channels are unreliable. If *dc* is not empty, the top element is dropped (*d-drp*) or duplicated (*d-dup*), and if *ac* is not empty, the top element is dropped (*a-drp*) or duplicated (*a-dup*). Fig. 6 shows a graphical representation of a state of ABP.

Each state of ABP is formalized as a term $\{(sb : b_1)$ $(data : d(n))$ $(rb : b_2)$ $(buf : dl)$ $(dc : q_1)$ $(ac : q_2)\}$, where b_1 and b_2 are Boolean values, *n* is a natural number, *dl* is a data list, q_1 is a queue of pairs of data and Boolean values and q_2 is a queue of Boolean values. $d(n)$ denotes data to be delivered from the sender to the receiver. Initially, b_1 is *true*, b_2 is *true*, *n* is 0, *dl* is the empty list, q_1 is the empty queue, and q_2 is the empty queue. The state transitions that formalize the actions are specified in rewrite rules as follows:

```
rl  [d-snd] : {(sb: B)(data: D)(dc: Ps) OCs}
=> {(sb: B)(data: D)(dc:(Ps | < D,B >)) OCs}.
crl [a-rec1] : {(sb: B)(data: d(N))
  (ac: (B' | Bs)) OCs}
=> {(sb:(not B))(data: d(N + 1))(ac: Bs) OCs}
if B =/= B' .
crl [a-rec2] : {(sb: B)(data: D)
  (ac: (B' | Bs)) OCs}
=> {(sb: B)(data: D)(ac: Bs) OCs} if B = B'.
rl  [a-snd] : {(rb: B)(ac: Bs) OCs}
=> {(rb: B) (ac: (Bs | B)) OCs} .
crl [d-rec1] : {(rb: B)(buf: Ds)
  (dc: (< D,B' > | Ps)) OCs}
=> {(rb: (not B))(buf: (Ds | D))(dc: Ps) OCs}
if B = B' .
crl [d-rec2] : {(rb: B)(buf: Ds)
  (dc: (< D,B' > | Ps)) OCs}
=> {(rb: B)(buf: Ds)(dc: Ps) OCs}
if B =/= B' .
rl  [d-drp] : {(dc: (Ps1 | P | Ps2)) OCs}
=> {(dc: (Ps1 | Ps2)) OCs} .
rl  [d-dup] : {(dc: (Ps1 | P | Ps2)) OCs}
=> {(dc: (Ps1 | P | P | Ps2)) OCs} .
rl  [a-drp] : {(ac: (Bs1 | B | Bs2)) OCs}
=> {(ac: (Bs1 | Bs2)) OCs} .
rl  [a-dup] : {(ac: (Bs1 | B | Bs2)) OCs}
=> {(ac: (Bs1 | B | B | Bs2)) OCs} .
```

Words that start with a capital letter, such as B, D, Ps, and OCs, are Maude variables. B, D, Ps, and OCs are variables of Boolean values, data, queues of (Data,Bool)-pairs, and observable component soups, respectively. The types (or sorts) of the other variables can be understood from what have been described. The two rewrite rules `a-rec1` and `a-rec2` formalize action *a-rec*. What rewrite rules formalize what

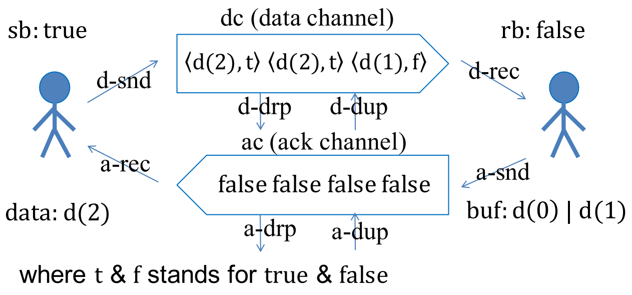

FIGURE 6. A state of ABP.

actions can be understood from what have been described. Let S_{ABP} refer to the specification of ABP in Maude. A concurrent program P_{ABP} is written in Java based on S_{ABP}, where one thread performs two actions *d-snd* and *a-rec*, one thread performs two actions *a-snd* and *d-rec*, one thread performs two actions *d-drp* and *a-drp*, and one thread performs two actions *d-dup* and *a-dup*. We intentionally insert one flaw in P_{ABP} such that when the receiver gets the third data, it does not put the third data into *buf* but puts the fourth data into *buf*.

2) EXPERIMENT

Algorithm 2 shows the pseudo-code for sender, receiver, dropper, and duplicator actions that are executed simultaneously by multiple threads in the program. Actions *d-send* and *a-rec* are implemented by the code fragments at line 6 and lines 9 - 13, respectively. Actions *a-send* and *d-rec* are implemented by the code fragments at line 20 and lines 23 - 27, respectively. Actions *a-drp* and *d-drp* are implemented by the code fragments at lines 32 and 35, respectively. Actions *a-dup* and *d-dup* are implemented by the code fragments at lines 40 and 43, respectively. $lock_{dc}$ and $lock_{ac}$ are two locks that are used to protect those atomic code fragments. Note that $lock_{dc}$ may be the same as $lock_{ac}$ when one lock is actually used. We can see that one atomic fragment protected by $lock_{dc}$ and another atomic fragment protected by $lock_{ac}$ can be executed in parallel. Hence, at most two atomic code fragments may be executed by multiple threads in parallel (or simultaneously), although each atomic action (or state transition) in the formal specification is implemented by one atomic code fragment in the program.

We suppose that the sender is to deliver four data to the receiver, the depth of each layer is 100, and *DEPTH* is unbounded. The simulation relation candidate from P_{ABP} to S_{ABP} is essentially the identify function. We change each channel size as follows: one, two, and three. We do not need to fix the number of layers in advance, but the number of layers can be determined by the tool on the fly. For each experiment, however, the number of layers is larger than two. Table 1 shows experimental data in which one lock is used in the program, meaning that the maximum number of transition steps is one, while Table 2 shows experimental data in which two locks are used in the program, meaning that the maximum number of transition steps is two. We can change Algorithm 2

Algorithm 2: Sender, Receiver, Dropper, and Duplicator in ABP Program

```
input : $ABP$ – a concurrent program $lock_{dc}$ – a lock on
        the data channel $lock_{ack}$ – a lock on the ack
        channel
$dc \leftarrow empty$; $ac \leftarrow empty$;
function $Sender$ is
    $data \leftarrow 0$; $sb \leftarrow true$;
    while $true$ do
        $request(lock_{dc})$;
        $dc.put(< data, sb >)$;
        $release(lock_{dc})$;
        $request(lock_{ac})$;
        if $ac.size() > 0$ then
            $b \leftarrow ac.get()$;
            if $b \neq sb$ then
                $sb \leftarrow \neg sb$;
                $data \leftarrow data + 1$;
        $release(lock_{ac})$;
function $Receiver$ is
    $buf \leftarrow 0$;
    $rb \leftarrow true$;
    while $true$ do
        $request(lock_{ac})$;
        $ac.put(rb)$;
        $release(lock_{ac})$;
        $request(lock_{dc})$;
        if $dc.size() > 0$ then
            $< d, b > \leftarrow dc.get()$;
            if $b = rb$ then
                $buf.put(d)$;
                $rb \leftarrow \neg rb$;
        $release(lock_{dc})$;
function $Dropper$ is
    while $true$ do
        $request(lock_{ac})$;
        $ac.get()$;
        $release(lock_{ac})$;
        $request(lock_{dc})$;
        $dc.get()$;
        $release(lock_{dc})$;
function $Duplicator$ is
    while $true$ do
        $request(lock_{ac})$;
        $ac.duptop()$;
        $release(lock_{ac})$;
        $request(lock_{dc})$;
        $dc.duptop()$;
        $release(lock_{dc})$;
```

TABLE 1. Experimental data for ABP program in which one lock is used.

Channel size	Worker	Time (d:h:m)	#seqs
1	Worker 1	0:5:33	47,505
	Worker 2	0:5:31	
	Worker 3	0:5:47	
	Worker 4	0:5:47	
2	Worker 1	5:23:27	4,606,719
	Worker 2	6:0:32	
	Worker 3	5:23:55	
	Worker 4	5:23:26	
3	Worker 1	33:17:26	37,403,548
	Worker 2	33:12:11	
	Worker 3	33:15:24	
	Worker 4	33:15:06	

TABLE 2. Experimental data for ABP program in which two locks are used.

Channel size	Worker	Time (d:h:m)	#seqs
1	Worker 1	0:8:14	64,854
	Worker 2	0:8:31	
	Worker 3	0:8:14	
	Worker 4	0:8:39	
2	Worker 1	7:13:21	6,611,839
	Worker 2	7:12:30	
	Worker 3	7:11:58	
	Worker 4	7:12:15	
3	Worker 1	38:16:12	54,429,058
	Worker 2	38:16:41	
	Worker 3	38:14:11	
	Worker 4	38:21:34	

to make the program using only one lock easily by replacing two locks $lock_{dc}$ and $lock_{ac}$ with the same lock as mentioned.

When each channel size was one, one lock experiment took about 5 hours 47 minutes, while two locks experiment took about 8 hours 39 minutes to generate all state sequences with four workers and check them with Maude on the fly. The number of the state sequences generated is 47,505 and 64,854, respectively. Note that the number of state sequences is the total number of sub-state sequences at each layer without combining sub-state sequences. Maude detected that some state sequences have adjacent states s and s' such that s cannot reach s' by S_{ABP} in both experiments with one and two state transitions, respectively. If that is the case, a tool component [13] implemented in Maude shows us some information as follows:

```
Result4Driver?: {seq: 31,msg: "Failure",
from: {sb: true data: d(2) rb: true buf:
   (d(0) | d(1)) dc: < d(2),true > ac: nil},
to:{sb: true data: d(2) rb: false buf:
   (d(0) | d(1) | d(3)) dc: nil ac: nil},
index: 3,bound: 2}
```

This is because although the receiver must put the third data `d(2)` into *buf* when `d(2)` is delivered to the receiver, the receiver instead puts the fourth data `d(3)` into *buf*, which is the flaw intentionally inserted into P_{ABP}. This demonstrates that our tool can detect the flaw.

When each channel size was two, one lock experiment took about 6 days, while two locks experiment took about 7 days 13 hours 21 minutes to generate all state sequences with four workers and check them with Maude on the fly. The number of the state sequences generated is 4,606,719 and 6,611,839, respectively. As is the case in which each channel is one, Maude detected that some state sequences have adjacent states s and s' such that s cannot reach s' by S_{ABP} in both experiments with one and two state transitions, respectively, due to the flaw intentionally inserted in P_{ABP}.

When each channel size was three, one lock experiment took about 33 days 17 hours 26 minutes, while two locks experiment took about 38 days 21 hours 34 minutes to generate all state sequences with four workers and check them with Maude on the fly. The number of the state sequences generated is 37,403,548 and 54,429,058, respectively. As is the case in which each channel is one and two, Maude detected that some state sequences have adjacent states s and s' such that s cannot reach s' by S_{ABP} in both experiments with one and two state transitions, respectively, due to the flaw intentionally inserted in P_{ABP}.

The experimental data in Tables 1–2 show that the more locks are used, the more time it takes to do testing programs. It is reasonable because the more locks used introduce more synchronized points in programs from which more state sequences are generated. In addition, our experiments also demonstrate that programmers are able to know the maximum steps in programs for one atomic state transition in formal specifications based on the number of locks used in programs.

If we did not use the proposed approach and simply used JPF to generate state sequences with the same computer, we encountered an out-of-memory error even when each channel size was one [14]. It was reported [26] that when each channel was three and the number of data delivered was three (but not four), a straightforward use of JPF did not complete a model checking experiment for ABP into which no flaw was intentionally inserted, leading to an out-of-memory error after it took about four days with almost the same computer used in the experiments reported in the present paper. Therefore, the proposed technique can alleviate the out-of-memory situation due to the state space explosion.

We would like to demonstrate further the effectiveness of our parallel specification-based testing for testing concurrent programs by conducting more experiments for ABP case study with our divide & conquer approach to testing concurrent programs with different numbers of workers shown in Table 3. Even when the number of workers is one, our tool successfully completes the experiment, meaning that it alleviates the state space explosion, while the straightforward use of JPF did not as above-mentioned. We use a MacPro computer that carries a 2.5 GHz microprocessor with 28 cores and 1.5 TB memory to conduct the experiments because we need to use many workers. For ABP whose number of locks used is one, it takes 16 hours 40 minutes, 5 hours 4 minutes, 2 hours 50 minutes, and 2 hours 7 minutes to complete generating state sequences and checking them with Maude when the number of workers is one, four, eight, and twelve, respectively. Meanwhile, for ABP whose number of locks used is two, it takes 1 day 24 minutes, 7 hours 37 minutes, 4 hours 27 minutes, and 3 hours 13 minutes to complete generating state sequences and checking them with Maude when the number of workers is one, four, eight, and twelve, respectively. We plot the experimental data in Table 3 on the graph shown in Fig. 7. We can see that the verification time improves quickly for ABP programs when we increase the number of workers from one to four, which demonstrates that parallelization is effective for our proposed technique. The improvement for ABP with one and two locks used is about 69.9% and 68.8%, respectively. When we increase the number of workers from four to eight and eight to twelve, the verification time improves as well, although the improvement is slower. This is because the more workers used, the busier the master and the Redis cache need to handle and communicate with workers. Therefore, depending on the power of the machine used to conduct experiments, we may choose a reasonable number of workers when using our tool. In conclusion, we have demonstrated the power of our parallel specification-based testing for concurrent programs.

TABLE 3. Experimental data for ABP program with various numbers of workers.

Channel size	#locks	#seqs	#workers	Time (d:h:m)
1	1	47,505	1	0:16:40
			4	0:5:4
			8	0:2:50
			12	0:2:7
	2	64,854	1	1:0:24
			4	0:7:37
			8	0:4:27
			12	0:3:13

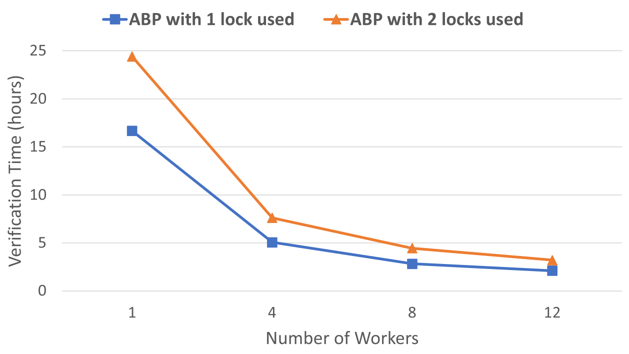

FIGURE 7. Verification time for ABP programs with various numbers of workers.

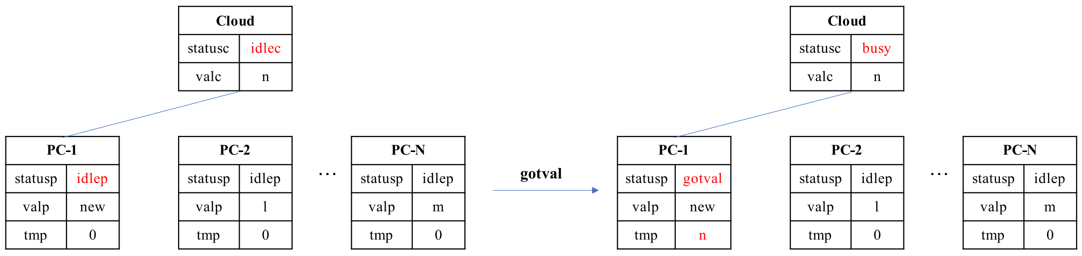

FIGURE 8. Gotval transition.

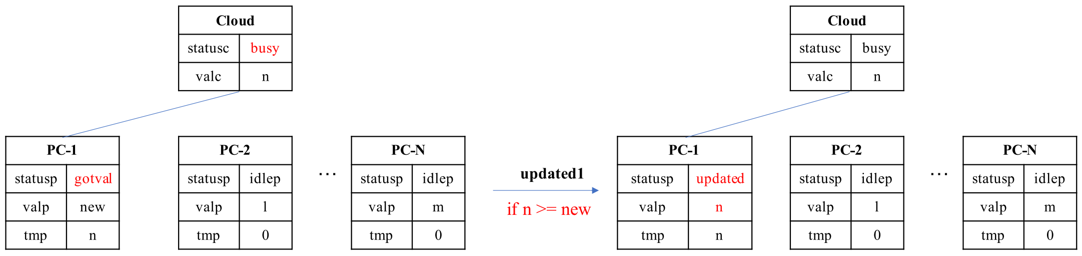

FIGURE 9. Updated1 transition.

B. REVISED CLOUDSYNC

1) INTRODUCTION

Revised CloudSync is a simplified cloud synchronization protocol in which many PCs would like to exchange messages with a *Cloud* in order. For simplicity, we use natural numbers as messages. A *PC* may connect to the *Cloud* if and only if both the *PC* and *Cloud* are in an idle state. After connected, the *PC* can fetch the value from the *Cloud* and then update either the value of *Cloud* or the value of the *PC* depending on which value is larger. The following shows how the Revised CloudSync protocol works in detail.

The *Cloud* maintains two pieces of information: *statusc* and *valc* that are the status and value of the *Cloud*, respectively. *statusc* is set to one of *idlec* and *busy* that are labels and *valc* is set to a natural number. *statusc* and *valc* are initially set to *idlec* and a natural number n, respectively. Meanwhile, each *PC* maintains three pieces of information: *statusp*, *valp*, and *tmp* that are the status, value, and temporary value of the *PC*, respectively. *statusp* is set to one of *idlep*, *gotval*, and *updated* that are also labels, and *valp* and *tmp* are set to natural numbers. *statusp*, *val*, and *tmp* are initially set to *idlep*, a natural number, such as *new*, l, and m, and the natural number 0. Note that n, *new*, l, and m are arbitrary natural numbers used. The protocol uses three transition rules.

The first transition rule is *gotval* depicted in Fig. 8. A *PC* wants to connect to the *Cloud* if and only if the *statusc* of the *Cloud* is *idlec* and the *statusp* of the *PC* is *idlep*. If that is the case, the *PC* fetches the current *valc* of the *Cloud* and updates the *tmp* of the *PC* to the value fetched; the *statusp* of the *PC* is also updated to *gotval*, while the *statusc* of the *Cloud* is changed to *busy*.

The second transition rule is *updated*. If the *tmp* of the *PC* involved is equal or greater than the *valp* of the *PC*, *updated* conducts *update*1 depicted in Fig. 9. Otherwise, it conducts *update*2 depicted in Fig. 10. Both *update*1 and *update*2 change the *statusp* of the *PC* to *updated*. *update*1 changes the *valp* of the *PC* to n, which is the same as the *tmp* of the *PC* and the *valc* of the *Cloud*, and leaves the other values unchanged, while *update*2 changes both the *valc* of the *Cloud* and the *tmp* of the *PC* to *new* and leaves the other values unchanged. Basically, the *updated* rule guarantees that the *valp* of the *PC* and the *valc* of the *Cloud* maintain the same largest number between two of them after the rule has been applied.

The last transition rule is *gotoidle* depicted in Fig. 11. After the *updated* rule has been just carried out by a *PC* and the *Cloud*, the *statusc* of the *Cloud* is *busy*, the *statusp* of the *PC* is *updated* and the *valc* of the *Cloud* and the *valp* and *tmp* of the *PC* have a same value, say *new*. If so, the *gotoidle* rule updates the *statusc* of the *Cloud* back to *idlec*, the *statusp* of the *PC* back to *idlep* and the *tmp* of the *PC* back to 0. The last transition rule makes the *Cloud* as well as the *PC* free. From now on, the *Cloud* can freely connect to any *PC* for exchanging messages subsequently.

The three transition rules are specified in rewrite rules as follows:

```
rl [getval] : {(cloud: < idlec,CVal >)
    (pc[P]: < idlep,PVal,OldCVal >) OCs}
=> {(cloud: < busy,CVal >)
    (pc[P]: < gotval,PVal,CVal >) OCs} .
crl [update1] : {(cloud: < busy,CVal >)
    (pc[P]: < gotval,PVal,GotCVal >) OCs}
=> {(cloud: < busy,CVal >)
    (pc[P]: < updated,GotCVal,GotCVal >) OCs}
    if GotCVal >= PVal .
crl [update2] : {(cloud: < busy,CVal >)
    (pc[P]: < gotval,PVal,GotCVal >) OCs}
=> {(cloud: < busy,PVal >)
    (pc[P]: < updated,PVal,PVal >) OCs}
```

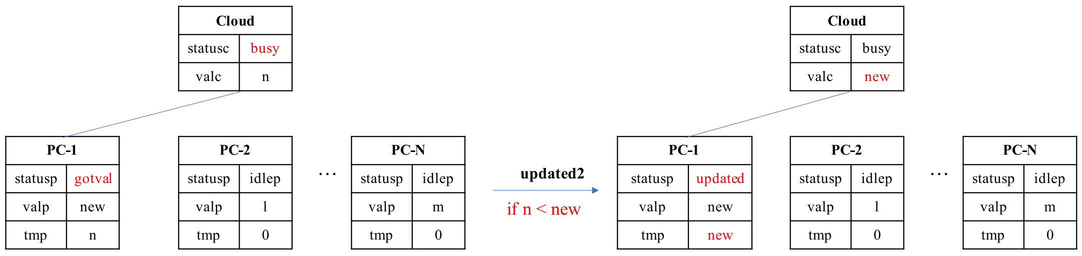

FIGURE 10. **Updated2 transition.**

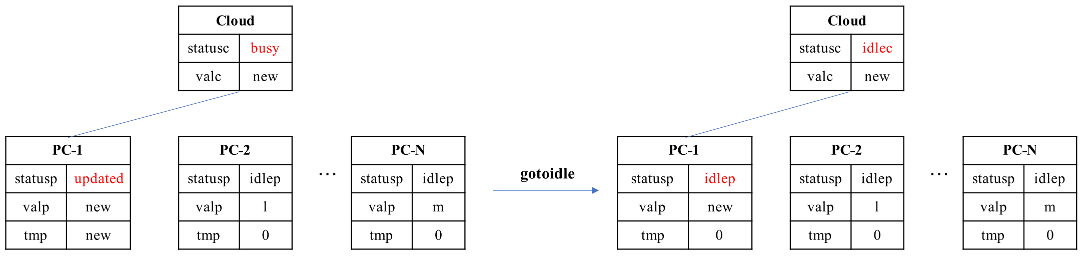

FIGURE 11. **Gotoidle transition.**

```
    if GotCVal < PVal .
rl [gotoidle] : {(cloud: < busy,CVal >)
    (pc[P]: < updated,PVal,OldCVal >) OCs}
=> {(cloud: < idlec,CVal >)
    (pc[P]: < idlep,PVal,0 >) OCs} .
```

Words starting with a capital letter, such as *P*, *PVal*, *GotCVal*, *CVal*, *OldCVal*, *RandVal*, and *OCs* are Maude variables. *P* and *OCs* are variables of *Pid* sort and observable component soups, respectively. *PVal*, *GotCVal*, *CVal*, *OldCVal*, and *RandVal* are variables of *Nat* sort. `idlec` and `busy` are the constants of *LabelC* sort that denote the possible status values of the *Cloud*. `idlep`, `gotval`, and `updated` are the constants of *LabelP* sort that denote the possible status values of a *PC*. How to implement CloudSync in Java is described in a document publicly available at Footnote 1, which resides under the *documents* folder.

2) EXPERIMENT

In the Revised CloudSync protocol experiment, three PCs and one Cloud are involved. Because the reachable state space is huge, we need to specify a depth bound to make sure that the experiments terminate. We conducted two experiments: (1) only one worker was used, *DEPTH* was 400 and the reachable state was not divided, and (2) four workers were used, the reachable state space was divided into two layers and each layer depth was 200 (namely that the global *DEPTH* was 400). The experimental data are shown in Table 4.

In these experiments, we did not intend to insert any bugs into the program. All state sequences generated by JPF workers were checked with Maude on the fly when the maximum number of transition steps was one and no bug was detected. For the experiment (1), it took 1 day 23 hours and 22 minutes to generate all state sequences and check them with Maude.

TABLE 4. **Experimental data for revised CloudSync.**

Depth	Worker	Time (d:h:m)	#seqs
400	One worker	1:23:22	4,449
400	Worker 1	0:1:36	3,118
	Worker 2	0:1:36	
	Worker 3	0:1:36	
	Worker 4	0:1:33	

The number of the state sequences generated is 4,449. For the experiment (2), it took 1 hour 36 minutes to generate all state sequences and check them with Maude. The number of the state sequences generated is 3,118. The experimental results show that (2) outperforms (1) and is 29 times faster than (1).

C. NSLPK

1) INTRODUCTION

Needham-Schroeder Public-Key authentication protocol (NSPK) can be described as three message exchanges:

Challenge: A $\rightarrow$ B : $\{N_a, A\}_{K_b}$
Response: B $\rightarrow$ A : $\{N_a, N_b\}_{K_a}$
Confirmation: A $\rightarrow$ B : $\{N_b\}_{K_b}$

where *A* and *B* are principals called an initiator and a responder, respectively, K_p is the public key owned by a principal p, N_p is a nonce generated by p, and m_{K_p} is the ciphertext obtained by encrypting a message m with K_p. Note that m_{K_p} can only be decrypted by a principal who owns the private key that corresponds to K_p. Lowe found an attack to NSPK and corrected it [27]. The corrected version is called NSLPK that can be described as follows:

Challenge: A $\rightarrow$ B : $\{N_a, A\}_{K_b}$
Response: B $\rightarrow$ A : $\{N_a, N_b, B\}_{K_a}$
Confirmation: A $\rightarrow$ B : $\{N_b\}_{K_b}$

The difference between NSPK and NSLPK is that the sender principal ID B is used to construct the *Response* message. The ciphertext obtained by encrypting N_a, N_b and B with the A's public key K_a. Let us describe the formal specification of NSLPK in Maude. We use the following operators as the constructors of observable components:

```
op nw:_ : Soup{Msg} -> OCom [ctor] .
op~rands:_ : Soup{Rand} -> OCom [ctor] .
op~nonces:_ : Soup{Nonce} -> OCom [ctor] .
op~prins:_ : Soup{Prin} -> OCom [ctor] .
```

where `Soup{Msg}`, `Soup{Rand}`, `Soup{Nonce}`, and `Soup{Prin}` are the sorts for soups of messages, random numbers, nonces, and principals, respectively. The `nw` observable component stores all messages sent by principals. The `rands` observable component stores the random numbers available. The `nonces` observable component stores the nonces gleaned by the intruder. The `prins` observable component stores the principals participating in the protocol. The `nw` observable component formalizes the network. We suppose that the network is initially empty and then the `nw` observable component is initially the empty soup denoted *emp*. We also suppose that there are two random numbers initially available, three principals (two trustable ones and one intruder), the `rands` observable component is initially $r1$ $r2$, and the `prins` observable component is initially p q *intrdr*, where p and q denote the two trustable principals and *intrdr* denotes the intruder. Because nothing has been initially gleaned by the intruder, the `nonces` observable component is initially *emp*. The initial state denoted `init` is as follows:

```
op init : -> Config .
eq~init = {(nw: emp) (rands: (r1 r2))
     (nonces: emp) (prins: (p q intrdr))} .
```

The message exchanges exactly obeying the protocol are specified in rewrite rules as follows:

```
rl [Challenge] : {(nw: NW) (nonces: Ns)
 (rands: (R Rs)) (prins: (P Q Ps))}
=> {(nw: (m1(P,P,Q,c1(Q,n(P,Q,R),P)) NW))
 (nonces: (if Q == intrdr then (n(P,Q,R) Ns)
   else Ns fi))
 (rands: Rs) (prins: (P Q Ps))} .
rl [Response] :
 {(nw: (m1(P',P,Q,c1(Q,N,P)) NW))
 (rands: (R Rs)) (nonces: Ns) OCs}
=> {(nw: (m2(Q,Q,P,c2(P,N,n(Q,P,R),Q))
 m1(P',P,Q,c1(Q,N,P)) NW)) (rands: Rs)
 (nonces: (if P == intrdr then
    (N n(Q,P,R) Ns) else Ns fi)) OCs} .
rl [Confirmation] :
 {(nw: (m2(Q',Q,P,c2(P,N,N',Q))
 m1(P,P,Q,c1(Q,N,P)) NW)) (nonces: Ns) OCs}
=> {(nw: (m3(P,P,Q,c3(Q,N'))
 m2(Q',Q,P,c2(P,N,N',Q))
 m1(P,P,Q,c1(Q,N,P)) NW))
 (nonces: (if Q == intrdr then (N' Ns)
   else Ns fi)) OCs} .
```

Messages are formalized in the form $mi(P', P, Q, C)$ for $i = 1, 2, 3$, where P' is the actual sender, P is the seeming sender, Q is the receiver, and C is a ciphertext. P' cannot be seen by principals. If P' is different from P, then P' must be *intrdr* and the message has been forged by the intruder. For example, rewrite rule `Response` says that if there exists a *Challenge* message $m1(P', P, Q, c1(Q, N, P))$ in the network, where $c(Q, N, P)$ denotes the ciphertext obtained by encrypting N and P with the Q's public key, and a random number R is available, then Q replies to the message by putting $m2(Q, Q, P, c2(P, N, n(Q, P, R), Q))$ into the network and the nonce $n(Q, P, R)$ made by Q is gleaned by the intruder if P is the intruder. Note that messages are never deleted from the network. Faking messages by the intruder based on the nonces gleaned and the messages in the network are specified in rewrite rules as follows:

```
rl [fake11] : {(nw: NW) (nonces: (N Ns))
 (prins: (P Q Ps)) OCs}
=> {(nw: (m1(intrdr,P,Q,c1(Q,N,P)) NW))
 (nonces: (N Ns)) (prins: (P Q Ps)) OCs} .
rl [fake12] : {(nw: (m1(P',P'',Q'',C1)
 NW)) (prins: (P Q Ps)) OCs}
=> {(nw: (m1(intrdr,P,Q,C1) m1(P',P'',Q'',C1)
 NW)) (prins: (P Q Ps)) OCs} .
rl [fake21] : {(nw: NW) (nonces: (N N' Ns))
 (prins: (P Q Ps)) OCs}
=> {(nw: (m2(intrdr,Q,P,c2(P,N,N',Q)) NW))
 (nonces: (N N' Ns)) (prins: (P Q Ps)) OCs} .
rl [fake22] : {(nw: (m2(Q',Q'',P'',C2) NW))
 (prins: (P Q Ps)) OCs}
=> {(nw: (m2(intrdr,Q,P,C2) m2(Q',Q'',P'',C2)
 NW)) (prins: (P Q Ps)) OCs} .
rl [fake31] : {(nw: NW) (nonces: (N Ns))
 (prins: (P Q Ps)) OCs}
=> {(nw: (m3(intrdr,P,Q,c3(Q,N)) NW))
 (nonces: (N Ns)) (prins: (P Q Ps)) OCs} .
rl [fake32] : {(nw: (m3(P',P'',Q'',C3) NW))
 (prins: (P Q Ps)) OCs}
=> {(nw: (m3(intrdr,P,Q,C3) m3(P',P'',Q'',C3)
 NW)) (prins: (P Q Ps)) OCs} .
```

For example, rewrite rule `fake21` says that if there are two nonces N and N' gleaned by the intruder, the intruder fakes $m2(intrdr, Q, P, c2(P, N, N', Q))$; rewrite rule `fake22` says that if there is $m2(Q', Q'', P'', C2)$ in the network, the intruder fakes $m2(intrdr, Q, P, C2)$, where P and Q are principals chosen randomly. How to implement NSLPK in Java is described in a document publicly available at Footnote 1, which resides under the *documents* folder.

2) EXPERIMENT

In the experiments, we suppose that there are two non-intruder principals, one intruder, and two random numbers. Because the state space is huge, a bounded depth is used to generate state sequences to make sure that the experiments terminate. We conduct two experiments: (1) only one worker was used, *DEPTH* was 200 and the reachable state was not divided, and (2) four workers were used, the reachable state space was divided into two layers and each layer depth was 100 (namely that the global *DEPTH* was 200). The experimental data are shown in Table 5.

In these experiments, we did not intend to insert any bugs into the program. All state sequences generated by JPF were checked with Maude on the fly when the maximum

TABLE 5. Experimental data for NSLPK.

Depth	Worker	Time (d:h:m)	#seqs
200	One worker	8:18:09	1,117,537
200	Worker 1	0:14:23	109,933
	Worker 2	0:14:21	
	Worker 3	0:14:35	
	Worker 4	0:14:32	

TABLE 6. Experimental data for original CloudSync.

Depth	Worker	Time (d:h:m)	#seqs
400	One worker	1:18:49	16,185
400	Worker 1	8:19:11	433,611
	Worker 2	8:18:24	
	Worker 3	8:17:41	
	Worker 4	8:17:34	

number of transition steps was one and no bug was detected. For the experiment (1), it took over 8 days to generate all state sequences and check them with Maude. The number of the state sequences generated is 1,117,537. For the experiment (2), it took about 14 hours to generate all state sequences and check them with Maude. The number of the state sequences generated is 109,933. The experimental results show that (2) outperforms (1) and is about 14 times faster than (1).

We conduct one more experiment for NSLPK with the same configuration as the experiment (2) above, however, a bug is intentionally inserted into the program in which the sender information is not included in the *Response* messages from principals. Our tool can quickly detect the bug just in some seconds and show a warning message as follows:

```
warning ({nw: emp rand: (r1 r2) nonces: emp
  prins: (p q intrdr)} |
{nw: (m1(p,p,q,c1(q,n(p,q,r1),p))) rand: (r2)
  nonces: emp prins: (p q intrdr)} |
{nw: (m1(p,p,q,c1(q,n(p,q,r1),p))
  m2(q,q,p,c2(p,n(p,q,r1),n(q,p,r2))))
  rand: emp nonces: emp prins: (p q intrdr)}
  | nil)
```

This is because the message *m*2, a *Response* message, in the network *nw* does not consist of the sender information in the ciphertext *c*2. Hence, Maude cannot parse the message and show the warning message. This demonstrates that our tool can detect the flaw.

D. ORIGINAL CLOUDSYNC

1) INTRODUCTION

Original CloudSync is the original version of CloudSync protocol that is the same as Revised CloudSync except for some points, which then are described in this sub-section. For convenience, Original CloudSync is called as CloudSync. As described above, Revised CloudSync uses three transition rules that are *gotval*, *updated*, and *gotoidle*. CloudSync uses the same three transition rules described above and one more transition rule: *modval* that is depicted in Fig. 12.

The *modval* transition rule does not care about the initial *valp* of PCs. Before exchanging messages between the *Cloud* and a *PC*, the *statusc* of the *Cloud* is *idlec* and the *statusp* of the *PC* is *idlep*. The *modval* rule updates the *valp* of the *PC* to a random natural number, say *new*, meaning that the *PC* is willing to be connected to the *Cloud*. For simplicity, the *new* value is the current value of *valp* plus one in our specification. The *modval* transition rule is specified in the following rewrite rule:

```
rl [modvalue] :
{(pc[P]: <idlep,PVal,OldCVal>) OCs} =>
{(pc[P]: < idlep,s(PVal),OldCVal >) OCs} .
```

where `s(PVal)` denotes $PVal + 1$. How to implement Original CloudSync in Java is described in a document publicly available at Footnote 1, which resides under the *documents* folder.

2) EXPERIMENT

In the CloudSync case study, three PCs and one Cloud are involved. A depth bound is used to make sure that the experiments terminate while generating state sequences due to the huge state space. We conduct two experiments: (1) only one worker was used, *DEPTH* was 400 and the reachable state was not divided, and (2) four workers were used, the reachable state space was divided into two layers and each layer depth was 200 (namely that the global *DEPTH* was 400). The experimental data are shown in Table 6.

In these experiments, we did not intend to insert any bugs into the program. All state sequences generated by JPF were checked with Maude on the fly when the maximum number of transition steps was one and no bug was detected. our approach. For the experiment (1), it took 1 day 18 hours and 49 minutes to generate all state sequences and check such state sequences with Maude. The number of the state sequences was 16,185. For the experiment (2), it took more than 8 days 17 hours to generate all state sequences and check the state sequences with Maude. The number of the state sequences was 433,611. The experimental results show that (1) outperforms (2), meaning that the straightforward use of JPF works effectively than our tool in this case study.

In CloudSync, whenever *modval* is used, the *valp* of a *PC* being to be connected with the *Cloud* is increased by one before the *PC* and the *Cloud* are connected. Therefore, the reachable state space of CloundSync becomes bigger than that of Revised CloudSync because the *modval* rule produces many different states. It implies that there may be many states located at the bottom of each layer. Furthermore, each *PC* has an equal chance to connect with the *Cloud* and only one *PC* can connect with the *Cloud* at one time. Therefore, the increment of *valp* makes the values of *PCs* turn over and so there may be some cross transitions between states in different sub-state spaces in a layer or even some backward transitions from

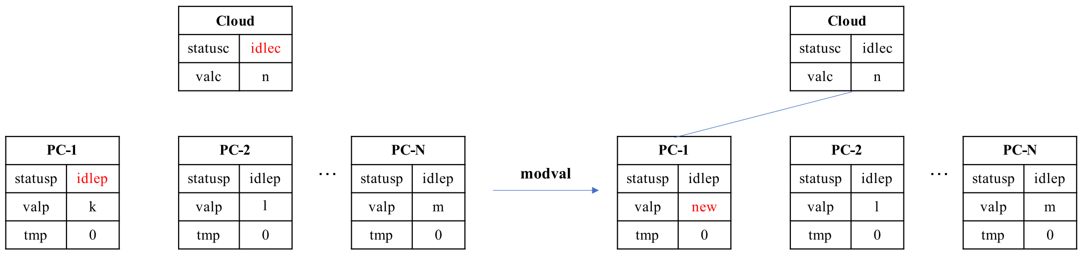

FIGURE 12. Modval transition.

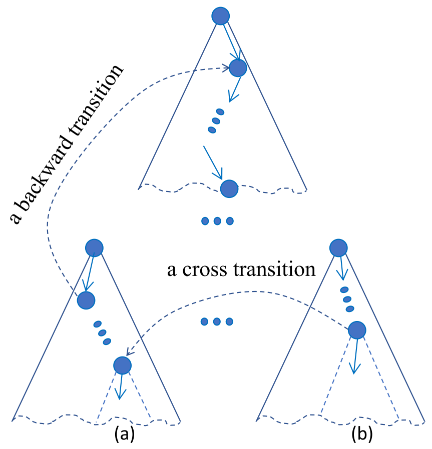

FIGURE 13. Backward and cross transitions.

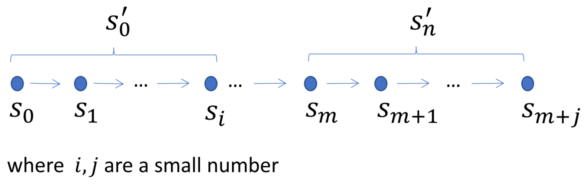

FIGURE 14. A state sequence in a layer *l*.

some states in sub-state spaces in a layer to some states in sub-state spaces in previous layers (see Fig. 13). We suppose that there are two states located in two different sub-state spaces in a layer such that there is a cross transition between them as shown in Fig. 13. The two sets ((a) and (b) shown in Fig. 13) of states located at the bottom of the layer reachable from the two states are different because the depths of the states are different. We only store states located at each layer in a big cache to remove duplicate states, but do not store all states in the reachable state space due to the state space explosion problem. Therefore, there are many states being collected at the layer although there are many states shared by many sub-state spaces that start from those collected states. Because of this situation, it makes our environment inefficient when the workers need to explore many states shared by many sub-state spaces.

It is necessary to make each sub-state space much smaller than the original reachable state space so as to use our tool effectively. If a system under testing has some long lasso loops, there may have many states shared by sub-state spaces at each layer, making many duplicated works for workers. Thus, we should avoid any long lasso loops under a system under test. In general, it would not be straightforward to get rid of all long lasso loops from a system under test. Thus, we need to come up with a technique that can handle such long loops, which is one piece of our future work.

E. THREATS TO BUG DETECTION

As described, we divide the reachable state space from each initial state to multiple layers, generating multiple sub-state spaces. For each layer l, given a state s_0, JPF instance needs to explore the sub-state space reachable from s_0 up to the layer depth $d(l)$ to generate its sub-state sequences. We suppose that there is a state sequence represented in JPF in the layer l that starts from s_0 as shown in Fig. 14. As described above, whenever JPF visits a state, we need to extract the values of observable components from the program under test by looking inside the heap of JPF, and then construct a state representation in the form of state expressions used in a specification. Because the execution units in a program are much finer than those in a specification, and therefore there are some consecutive states in a state sequence represented in JPF that have the same observable component values and so we construct the same state represented in a specification. For example, from s_0 to s_i, they have the same observable component values, and so s'_0 is constructed (see Fig. 14). Similarly, from s_m to s_{m+j}, they have the same observable component values and so s'_n is constructed. We finally obtain the state sequence $s'_0 \ldots s'_n$ such that two consecutive states are different. The state sequence is much shorter than the state sequence $s_0 \ldots s_{m+j}$ represented in JPF and can be checked with Maude. If the last state s'_n does not exist in a cache of states, it is sent to RabbitMQ master (1) to distribute to a worker subsequently. Given s'_n, we may generate $s_{m+j}, \ldots, s_{m+j+k}$ or $s_m, \ldots, s_{m+j}, \ldots, s_{m+j+l}$ or something else represented in JPF, where $l < k$, and then this may affect the size of state sequences to be checked by our tool, where the size may be less than *DEPTH*. Therefore, if a bug locates nearly at the boundary of the bounded depth, there may be a case in which our tool may overlook the bug. One possible way to mitigate this problem is that we can increase the depth of the final layer so that the bug can be covered and detected.

VIII. RELATED WORK

The state space explosion is the main challenge to concurrent program verification due to inherent thread interleavings. Several techniques have recently been studied to overcome this problem, such as bounded model checking and abstraction refinement. Nevertheless, these existing techniques cannot overcome the problem reasonably well, especially for large concurrent program verification. To take full advantage of multiprocessor architectures, parallel processing is a main stream to deal with this problem.

Bounded model checking (BMC) is an efficient technique for sequential program analysis that uses a symbolic representation to formalize the program verification problem into an equisatisfiable conjunctive normal form (CNF) formula that can be analyzed by a SAT/SMT solver. A CNF formula is a conjunction of one or more clauses in which each clause is a disjunction of literals, where a literal is an atomic formula or its negation. Among existing studies for BMC, CBMC is a famous bounded model checker to verify sequential C programs [18]. Based on CBMC, a SAT-based bounded verification technique is then proposed to support multi-threaded C programs, called TCBMC [28]. For Java programs, JBMC is a bounded model checking tool for verifying Java bytecode [19], which is also based on CBMC. However, JBMC only supports sequential Java programs. They are currently extending JBMC to support multi-threaded Java programs. Another extension of SAT/SMT-based BMC to model check concurrent programs is Lazy Sequentialization (Lazy-CSeq) [29], [30]. Given a concurrent program P together with two parameters u and r that are the loop unwinding bound and the number of round-robin schedules, respectively, they first generate an intermediate bounded program P_u by unwinding all loops and inlining all function calls in P with u as a bound except for those used for creating threads. P_u then is transformed into a sequential program $Q_{u,r}$ that simulates all behaviors of P_u within r round-robin schedules. $Q_{u,r}$ is then transformed into a propositional formula that can be analyzed by a SAT/SMT solver. To take advantage of parallelization, they then propose a method [31] to decompose the set of execution traces of concurrent programs into symbolic subsets that can be separately explored by multiple instances of a SAT solver in parallel. Given a number of threads and a number of execution contexts, their approach needs to calculate symbolic partitioning in which a single formula is divided into multiple propositional sub-formulas. After that threads are spawned to check such sub-formulas by using SAT in parallel. They build a prototype tool that can be deployed on a single machine as well as in a distributed environment.

To use much larger compute and memory resources than those of one single ordinary computer, distributed bounded model checking is one promising way to make verification scalable. Prantik *et al.* [32] then propose an algorithm that dynamically unfolds the call graph of a program and frequently splits it into sub-tasks that can be solved by many instances of an SMT solver in parallel. In detail, the technique splits the set of program paths into disjoins subsets that are searched independently. The splitting is done by picking a control node (splitting node) and considering (1) the set of paths that go through the node, and (2) the set of paths that do not. They create multiple processes, each of which has access to the input program and are deployed in a distributed environment. One process is designated as the server while the rest are called clients. The search starts sequentially on one of the clients. A client chooses a splitting node from which two partitions are created. The client continues verification on one of the partitions and sends the other partition to the server. The server is in charge of collecting, prioritizing partitions from the clients, and distributing them to the clients subsequently. Note that clients can split multiple times. This process continues until a client reports a counterexample or there is no partition left in the server and all clients are idle. Their architecture is basically a master-worker model that is similar to the architecture used in our tool. However, the way they partition the reachable state space is different from our approach that splits the reachable state space into multiple smaller sub-state spaces based on depth information. In recent years, exploiting the power of GPU in parallel computation with a huge number of cores, some researchers have used GPU to speed up the model checking verification [33]–[35], which may be one of our interesting future directions to improve our approach.

About abstraction refinement, the scheduling constraint based abstraction refinement (SCAR) is an effective method for concurrent program verification [36]. SCAR is built on top of CBMC that is a bounded model checker for C and C++ programs. However, instead of using the scheduling constraint [37], SCAR ignores the scheduling constraint and uses a scheduling constraint based abstraction refinement method that makes the constraints in the initial abstraction to be reduced significantly. From the initial abstraction, they use graph-based algorithms over an event order graph (EOG) for counterexample validation. Given that abstraction, a counterexample π is produced, which is a set of assignments to the variables in the abstraction. To validate the feasibility of π, they validate its corresponding EOG, which captures all the order requirements among the events of π. Some order requirements deducible from the EOG are called derived orders of the EOG. The derived orders are produced based on three rules by looping and they try to apply each rule to the EOG in sequence until this process reaches a fixpoint where no derived order can be produced any more. From obtained derived orders, if there exists a cycle, then the EOG is infeasible. Otherwise, it is not sure whether the EOG is feasible or not. If the EOG is feasible, it is truly a counterexample. Whenever counterexample validation is determined to be infeasible, the abstraction is refined by a refinement generation method that obtains a set of constraints and then can be encoded into simple constraints and reduce a large amount of space.

To take advantage of the SCAR method, a parallel refinement is proposed in which multiple engines are used to refine the abstraction simultaneously [38]. Their parallel technique performs on the whole abstraction search space instead of dividing the search space into small ones. Each engine does three steps that iteratively select a counterexample from the abstraction, validate the counterexample (CE), and analyze the refinement constraint (RC). Because they perform on the whole reachable search space, to select a counterexample and avoid duplication, they use a random search strategy. Besides, all engines share the learnt clause lib that can be updated and checked whether a counterexample is verified or not by engines. All engines also use the same RC lib, whenever an engine generates a refinement constraint. The constraint is added to the RC lib that is simplified to avoid redundant constraints. By sharing RC lib, each engine can obtain multiple refinement constraints in each iteration that makes the number of required iterations for each engine reduced. For each iteration, if one engine returns UNSAT, then the property is proven safe, and finishes verification. Otherwise, in the case of SAT, a counterexample is returned to validate the feasibility. If the counterexample is feasible, all engines will terminate and *Unsafe* is returned. Otherwise, the engine keeps on doing to analyze RC. Their approach is similar to our approach when they use the shared RC lib to avoid redundant constraint, while we use a shared cache to avoid redundant states and state sequences. However, our approach does not deal with the whole search space by each worker and so we do not need to consider the random search strategy and each worker conducts its sub-state spaces independently.

The existing parallel techniques mentioned above, where SAT/SMT solvers play the main role, are different from our approach. The advantage of their approach is that it is fully automatic to verify concurrent programs, while our approach starts from a specification and then implements a concurrent program based on the specification. Our technique can be used to verify not only a concurrent Java program [39], but also check the concurrent program that conforms to the specification, which can be used to complement the very last step in correct-by-constructions software development. We do not compare our tool with existing tools based on BMC and SCAR mentioned above because they are dedicated to verifying C and C++ programs. Besides, JBMC is used to model check sequential Java programs, but not concurrent Java programs. JPF is one of the most mature model checkers for Java programs that is why it is used to evaluate and compare with JBMC [19] in terms of correctness and running performance. Therefore, it is worth comparing our tool with JPF as well as improving the running performance of JPF.

In correct-by-construction system or software development, Ge *et al.*, [10] propose a High-Level Language (HLL) for Event-B that can be used between Event-B models and C code. Event-B models are translated to respective HLL models, where Event-B invariants are proved using a SAT-based model checker, from which C code is automatically generated. Although the authors claim that they propose a technique that makes it possible to conduct conformance proofs between HLL models and the C code generated from the models, they do not describe how they do so in detail. Our paper describes how to check the conformance of programs with specifications in detail that can be regarded as a complement to the very last step in the correct-by-construction technique.

Dalvandi *et al.* [2] propose a way to generate executable code in Dafny [40] from scheduled Event-B models. Scheduled Event-B is an augmented version of Even-B such that a scheduling language is used to make the control flow in an Event-B model explicit and facilitate derivation of algorithmic structure in Event-B refinement. Generated executable code can be verified with a static program verifier, such as Z3, because code is written in Dafny. Generated executable code is dedicated to sequential programs, while our tool can test concurrent programs.

Rivera *et al.* [3] propose a way to generate JML-annotated Java programs from Even-B models. They build a tool called EventB2Java to support the proposed program generation technique. Two case studies are conducted with EventB2Java to demonstrate the usefulness of the proposed technique and the support tool. Both sequential and concurrent programs can be generated with EventB2Java. Note that there is a sentence "…JML […] is designed to specify arbitrary sequential Java programs …," though, in the paper [4]. Because generated programs are annotated with JML [41], Java programs generated by EventB2Java can be verified. However, the paper [3] does not describe how to formally verify generated Java programs. To the best of our knowledge, JML focuses on the sequential behavior of Java programs, while extending JML to support concurrency is in progress [42]. Hence, checking, verifying or testing JML-annotated Java concurrent programs is still not matured.

Tran-Jørgensen *et al.* [4] propose a way to automatically generate JML-annotated Java programs from VDM models. They address the semantic differences between the contract-based elements of VDM-SL and JML and describe how to use dynamic JML assertion checks to ensure the consistency of VDM's subtypes. It looks like that JML-annotated Java programs generated from VDM models are only sequential.

IX. CONCLUSION

We have proposed a new testing technique for concurrent programs in a stratified way. The proposed technique could be processed naturally in parallel, which has been utilized by the tool supporting the technique. The experiments reported in the paper demonstrate that the proposed technique can mitigate the state space explosion problem and largely improve the timing performance for testing for all cases except for one, which cannot be achieved with the straightforward use of only one JPF instance. The tool supporting the technique is dedicated to Java programs. However, our technique can be applied to other programming languages provided that we have a model checker for the language concerned and can interact with the model checker as we have done with JPF.

The experiments reported in the present paper demonstrate that concurrent programs in which there are no long lasso loops can be effectively tackled with the proposed technique, while those in which there exist long lasso loops cannot. The present paper reported on totally four case studies. The three programs can be tackled well with our tool, while one cannot. Accordingly, a non-small number of concurrent programs would be likely to belong to the first group if not all. However, we need to conduct some more case studies in which some other concurrent programs will be tackled with our tool in order to make sure that the proposed technique and our tool supporting it can mitigate the state space explosion reasonably well.

Regarding bugs that can be found by our technique/tool, we want to address invariant properties [43] that hold in the whole reachable state spaces of programs. Currently, we only consider finite state sequences generated from programs, so we cannot detect any liveness property flaws. As one piece of our future work, we will use some semantics of temporal logics defined over finite state sequences [44], [45] and extend the technique/tool so that some liveness properties can be tested.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers who carefully read an earlier version of the paper and gave them valuable comments without which they were not able to complete the present paper.

REFERENCES

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

[2] M. Dalvandi, J. M. Butler, A. Rezazadeh, and A. S. Fathabadi, "Verifiable code generation from scheduled event-B models," in *Abstract State Machines, Alloy, B, TLA, VDM, and Z* (Lecture Notes in Computer Science), vol. 10817, M. J. Butler, A. Raschke, T. S. Hoang, and K. Reichl, Eds. Southampton, U.K.: Springer, Jun. 2018, pp. 234–248.

[3] V. Rivera, N. Cataño, T. Wahls, and C. Rueda, "Code generation for event-B," *Int. J. Softw. Tools Technol. Transf.*, vol. 19, no. 1, pp. 31–52, 2017.

[4] P. W. V. Tran-Jørgensen, P. G. Larsen, and G. T. Leavens, "Automated translation of VDM to JML-annotated Java," *Int. J. Softw. Tools Technol. Transf.*, vol. 20, no. 2, pp. 211–235, Apr. 2018.

[5] C. Jones, *Systematic Software Development Using VDM*. Upper Saddle River, NJ, USA: Prentice-Hall, Jan. 1990.

[6] J. Woodcock and J. Davies, *Using Z—Specification, Refinement, and Proof* (Prentice Hall International Series in Computer Science). Upper Saddle River, NJ, USA: Prentice-Hall, 1996.

[7] E. Börger, "The ASM refinement method," *Formal Aspects Comput.*, vol. 15, nos. 2–3, pp. 237–257, Nov. 2003.

[8] J.-R. Abrial, *The B-Book—Assigning Programs to Meanings*. Cambridge, U.K.: Cambridge Univ. Press, Jan. 2005.

[9] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: An open toolset for modelling and reasoning in event-B," *Int. J. Softw. Tools Technol. Transf.*, vol. 12, no. 6, pp. 447–466, Nov. 2010.

[10] N. Ge, A. Dieumegard, E. Jenn, and L. Voisin, "Correct-by-construction specification to verified code," *J. Softw., Evol. Process*, vol. 30, no. 10, p. e1959, Oct. 2018.

[11] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All About Maude* (Lecture Notes in Computer Science), vol. 4350. Berlin, Germany: Springer, 2007.

[12] 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.

[13] C. M. Do and K. Ogata, "Specification-based testing with simulation relations," in *Proc. 31st Int. Conf. Softw. Eng. Knowl. Eng. (SEKE)*, Jul. 2019, pp. 107–112.

[14] C. M. Do and K. Ogata, "A divide & conquer approach to testing concurrent Java programs with JPF and Maude," in *Structured Object-Oriented Formal Language and Method*, H. Miao, C. Tian, S. Liu, and Z. Duan, Eds. Cham, Switzerland: Springer, 2020, pp. 42–58.

[15] K. Ogata and K. Futatsugi, "Simulation-based verification for invariant properties in the OTS/CafeOBJ method," *Electron. Notes Theor. Comput. Sci.*, vol. 201, pp. 127–154, Mar. 2008.

[16] 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 *Proc. FM*, in Lecture Notes in Computer Science, vol. 5014. Berlin, Germany: Springer, 2008, pp. 425–429.

[17] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal Methods Syst. Des.*, vol. 19, no. 1, pp. 7–34, Jan. 2001.

[18] D. Kroening and M. Tautschnig, "CBMC—C bounded model checker," in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, Eds. Berlin, Germany: Springer, 2014, pp. 389–391.

[19] L. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtik, "JBMC: A bounded model checking tool for verifying Java bytecode," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham, Switzerland: Springer, 2018, pp. 183–190.

[20] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 446–455, Jun. 2007.

[21] M. F. Atig, A. Bouajjani, and S. Qadeer, "Contextbounded analysis for concurrent programs with dynamic creation of threads," in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Kowalewski and A. Philippou, Eds. Berlin, Germany: Springer, 2009, pp. 107–123.

[22] Open Source. (2009). *Redis*. Accessed: Feb. 7, 2022. [Online]. Available: https://redis.io/

[23] Open Source. (2007). *RabbitMQ*. Accessed: Feb. 7, 2022. [Online]. Available: https://www.rabbitmq.com/

[24] Open Source. (2001). *Apache Commons Lang*. Accessed: Feb. 7, 2022. [Online]. Available: https://commons.apache.org/proper/commons-lang/

[25] Open Source. (1995). *MySQL*. Accessed: Feb. 7, 2022. [Online]. Available: https://www.mysql.com/

[26] K. Ogata, "Model checking designs with CafeOBJ—A contrast with a software model checker," in *Proc. Workshop Formal Method Internet Mobile Things*, Shanghai, China, 2014. [Online]. Available: http://www.jaist.ac.jp/~ogata/slides/ECNU2014Nov27-28.pdf

[27] G. Lowe, "An attack on the Needham–Schroeder public-key authentication protocol," *Inf. Process. Lett.*, vol. 56, no. 3, pp. 131–133, Nov. 1995.

[28] I. Rabinovitz and O. Grumberg, "Bounded model checking of concurrent programs," in *Computer Aided Verification*, K. Etessami and S. K. Rajamani, Eds. Berlin, Germany: Springer, 2005, pp. 82–97.

[29] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato, "Bounded model checking of multi-threaded C programs via lazy sequentialization," in *Proc. 26th Int. Conf. Comput. Aided Verification*, in Lecture Notes in Computer Science, vol. 8559, A. Biere and R. Bloem, Eds. Vienna, Austria: Springer, Jul. 2014, pp. 585–602.

[30] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato, "Bounded verification of multi-threaded programs via lazy sequentialization," *ACM Trans. Program. Lang. Syst.*, vol. 44, no. 1, pp. 1–50, Mar. 2022.

[31] O. Inverso and C. Trubiani, "Parallel and distributed bounded model checking of multi-threaded programs," in *Proc. 25th ACM SIGPLAN Symp. Princ. Pract. Parallel Program. (PPoPP)*. New York, NY, USA: Association for Computing Machinery, Feb. 2020, pp. 202–216.

[32] P. Chatterjee, S. Roy, B. P. Diep, and A. Lal, "Distributed bounded model checking," *Formal Methods Syst. Des.*, Jan. 2022.

[33] T. Neele, A. Wijs, D. Bošnački, and J. V. D. Pol, "Partial-order reduction for GPU model checking," in *Automated Technology for Verification and Analysis*, C. Artho, A. Legay, and D. Peled, Eds. Cham, Switzerland: Springer, 2016, pp. 357–374.

[34] A. Wijs, T. Neele, and D. Bosnacki, "GPUexplore 2.0: Unleashing GPU explicit-state model checking," in *Proc. Int. Symp. Formal Methods*, vol. 9995, Nov. 2016, pp. 694–701.

[35] R. DeFrancisco, S. Cho, M. Ferdman, and A. S. Smolka, "Swarm model checking on the GPU," in *Model Checking Software*, F. Biondi, T. Given-Wilson, and A. Legay, Eds. Cham, Switzerland: Springer, 2019, pp. 94–113.

[36] L. Yin, W. Dong, W. Liu, and J. Wang, "Scheduling constraint based abstraction refinement for multi-threaded program verification," 2017, *arXiv:1708.08323*.

[37] J. Alglave, D. Kroening, and M. Tautschnig, "Partial orders for efficient bounded model checking of concurrent software," in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds. Berlin, Germany: Springer, 2013, pp. 141–157.

[38] L. Yin, W. Dong, W. Liu, and J. Wang, "Parallel refinement for multi-threaded program verification," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 643–653.

[39] C. M. Do and K. Ogata, "A divide & conquer approach to testing concurrent programs with JPF*," in *Proc. 27th Asia–Pacific Softw. Eng. Conf. (APSEC)*, Dec. 2020, pp. 356–364.

[40] K. Rustan and M. Leino, "Developing verified programs with Dafny," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds., San Francisco, CA, USA, May 2013, pp. 1488–1490.

[41] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *Int. J. Softw. Tools Technol. Transf.*, vol. 7, no. 3, pp. 212–232, Jun. 2005.

[42] E. Rodríguez, M. Dwyer, C. Flanagan, J. Hatcliff, G. T. Leavens, and Robby, "Extending JML for modular specification and verification of multi-threaded programs," in *Object-Oriented Programming*, A. P. Black, Ed. Berlin, Germany: Springer, 2005, pp. 551–576.

[43] K. Ogata and K. Futatsugi, "Simulation-based verification for invariant properties in the OTS/CafeOBJ method," *Electron. Notes Theor. Comput. Sci.*, vol. 201, pp. 127–154, Mar. 2008.

[44] A. Bauer, M. Leucker, and C. Schallhart, "Comparing LTL semantics for runtime verification," *J. Log. Comput.*, vol. 20, no. 3, pp. 651–674, Jun. 2010.

[45] G. D. Giacomo and Y. M. Vardi, "Linear temporal logic and linear dynamic logic on finite traces," in *Proc. 23rd Int. Joint Conf. Artif. Intell. (IJCAI)*, 2013, pp. 854–860.

CANH MINH DO received the B.S. degree in information technology from the National Economics University, in 2013, and the M.S. degree in information science from the Japan Advanced Institute of Science and Technology (JAIST), in 2019, where he is currently pursuing the Ph.D. degree.

He has been working on how to efficiently test large concurrent programs as his Ph.D. research.

KAZUHIRO OGATA received the B.S., M.S., and Ph.D. degrees in engineering from Keio University, in 1990, 1992, and 1995, respectively.

He is currently a Professor with the Japan Advanced Institute of Science and Technology (JAIST). His research interests include applications of formal methods to systems, such as distributed systems and security protocols.