Parallel stratified random testing for concurrent programs

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

***Abstract*—For a concurrent program in Java, the reachable state space from each initial state is divided into L layers such that each layer depth is D_l for $l = 1, \dots, L$. If the program is exhaustively tested for each layer and there are m states at depth $D_1 + \dots + D_l$, then there are m sub-state spaces in layer $l + 1$ and each of the m sub-state spaces is exhaustively tested. Instead of exhaustively testing all the m sub-state spaces, we use a percentage $\delta_l\%$ for each layer l, randomly select about $0.01 \times \delta_l \times m$ ones among the m states at depth $D_1 + \dots + D_l$ and test only about $0.01 \times \delta_l \times m$ sub-state spaces in layer $l + 1$. For a Java implementation of the NSPK authentication protocol, even though the number of layers was 2 and each layer depth was 100, it did not complete in 3 weeks to exhaustively test the concurrent program. On the other hand, if the number of layers was 3, each layer depth was 100 and the percentages for layer 1 & layer 2 were 0.05% (or 0.1%) & 0.05% (or 0.1%), respectively, it completed in 19h to randomly test the concurrent program.**

***Index Terms*—concurrent programs, JPF, Maude, parallel testing, random testing**

I. Introduction

It is extremely hard to test concurrent programs effectively and efficiently. This is because there are many processes or threads running in parallel, which requires us to take into account many possible combinations caused by interleaving processes or threads. We have been working on testing concurrent programs so as to make it more effective and efficient to test concurrent programs. We first proposed specification-based testing with simulation relations [1] and then a divide & conquer approach [2] to testing concurrent Java programs with JPF [3] and Maude [4]. The divide & conquer approach to testing concurrent programs can be naturally parallelized. The reachable state space of a concurrent program is divided into multiple smaller sub-state spaces that can be tested in parallel. The technique proposed in [2] basically checks if each execution sequence generated from a Java concurrent program can be accepted by a formal specification in Maude, where JPF is used to generate execution sequences from a Java concurrent program and Maude is used to check if the execution sequences can be accepted by a Maude formal specification. We have also demonstrated that the technique proposed in [2] can be used to test Java concurrent programs without checking if execution sequences generated from Java programs can be accepted by Maude specifications [5] in which we demonstrate that we quickly detect a state of a Java implementation of the NSPK authentication protocol [6] where the nonce secrecy property is broken. However, we were not able to detect a state in which the one-to-many agreement (authentication) property is broken. This is because a state in which the latter property is broken is located at a much deeper position than a state in which the former property is broken.

To aim at making it possible to detect a state in which the one-to-many agreement property is broken, we introduce random state selection in our testing technique for concurrent programs. We suppose that a system (or a protocol) is formally specified in Maude and a concurrent program is written in Java based on the formal specification. To test such a concurrent program, we divide the reachable state space of the concurrent program into L layers such that each layer depth is D_l for $l = 1, \dots, L$. If there is one initial state in the concurrent program, there is one sub-state space in layer 1. If there are m states located at depth $D_1 + \dots + D_l$, there are m sub-state spaces in layer $l + 1$ if we do not use any random state selection. We use a percentage δ_l for $l = 1, \dots, L - 1$ to randomly select some states located at $D_1 + \dots + D_l$. If n_l states are generated at depth $D_1 + \dots + D_l$, approximately $0.01 \times \delta_l \times n_l$ states are randomly selected among the n_l states and then we have approximately $0.01 \times \delta_l \times n_l$ sub-state spaces in layer $l + 1$.

For a Java implementation of the NSPK authentication protocol, even though we used 2 layers and each layer depth was 100, it did not complete in three weeks to exhaustively test the concurrent program. On the other hand, when we used 3 layers, each layer depth was 100 and the percentages for layer 1 & layer 2 were 0.05% (or 0.1%) & 0.05% (or 0.1%), respectively, it completed in 19h to randomly test the concurrent program. Although we still did not find a state in which the one-to-many agreement property is broken, we made some progress toward making it possible to detect such a state.

The remaining part of the paper is organized as follows. Sect. II mentions some preliminaries. Sect. III describes how to generate states from Java concurrent programs with JPF. Sect. IV describes how to randomly generate states in a stratified way. Sect. V proposes parallel stratified random testing for concurrent programs. Sect. VI reports on a (preliminary) case study. Sect. VII mentions some existing related work. Sect. VIII finally concludes the paper and mentions some future directions.

978-1-7281-8915-4/20/$31.00 ©2020 IEEE
DOI 10.1109/QRS-C51114.2020.00024

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 $s \to_M s'$. 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 [4], a rewriting logic-based computer language, as a specification language because Maude makes it possible to use associative-commutative collections. State transitions are specified in Maude rewrite rules.

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

```
Loop : "RemainderSection(RS)"
  rs : repeat while test&set($lock$) = true;
    "CriticalSection(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* false 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* 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 $\{(\mathrm{lock} : b)\ (\mathrm{pc}[\mathrm{p1}] : l_1)\ (\mathrm{pc}[\mathrm{p2}] : l_2)\ (\mathrm{pc}[\mathrm{p3}] : l_3)\}$, where b is a Boolean value and each l_i is either rs or cs. 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`.

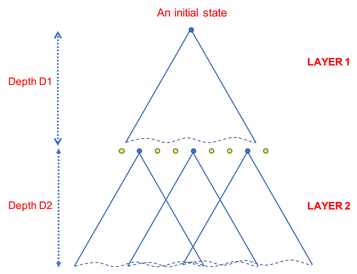

Fig. 1. Stratified random state generation

III. State 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 Java programs. It has a special Virtual Machine (VM) that can be backtracked to support model checking of concurrent Java programs so that it is able to detect some flaws lurking in programs, such as race conditions and deadlocks, when it reports a whole execution leading to the flaw. Basically, JPF can identify execution choices in a program from which the execution could proceed differently.

A state in JPF is mainly constituted of a heap and threads plus an execution history (or path) that leads to the state that is given a unique ID number. Looking inside the heap of a state, we may analyze the values of the data of a program at each state. JPF uses a search component in charge of selecting the next state from which the VM should proceed, either by directing the VM to generate the next state (forward) or by telling it to backtrack to a previously generated one (backtrack).

B. Generating states located at depth with JPF

The most important extension mechanism of JPF is listeners that allow us to observe, interact with, and extend JPF while executing. We can use the listener mechanism to analyze the values of the program's data as well as navigate JPF to transit between states. Indeed, we create a listener that subscribes to some events emitted from JPF execution. Whenever JPF hits to a depth bound, we look inside the heap of the state and extract all values of observer components from the program. Those values are then encapsulated into a Configuration object, then checking whether or not the object has already been explored before in a cache. If so, we do not need to take the object into account. Otherwise, we save the object into the cache and then send it to a message queue to handle later. Thereby, we could generate a set of states located at a given depth with JPF.

Because the state space could be enormous, we manage a $DEPTH$ bound parameter to make sure that JPF can terminate. The $DEPTH$ bound is the maximum depth from the initial state; once JPF reaches any state whose depth from the initial state is $DEPTH$, we send a backtrack message to request the search component for backtracking. The $DEPTH$ bound could be set unbounded, meaning that we ask JPF to check the entire state space. Note that JPF not only can generate states but also jointly verify programs that enjoy desired properties.

IV. Stratified random state generation

The use of JPF often encounters the notorious state space explosion while checking property violations through searching. Our previous work on a divide & conquer approach to testing concurrent programs can enhance the use of JPF in verification by parallelization [5]. In some cases, however, we were not able to complete testing of a concurrent program or to find a flaw located at a deep position in a concurrent program that has a huge reachable state space. For example, it is known that NSPK enjoys neither the nonce secrecy property nor the one-to-many agreement (or authentication) property and so does any of its implementations. A state in which the one-to-many agreement property is broken is located at a deeper position that one in which the nonce secrecy property is broken. A parallel version of the divide & conquer approach to testing concurrent program can detect a state in which the nonce secrecy property is broken lurking in an implementation of NSPK, while it cannot detect a state in which the one-to-many agreement property is broken. This is because even the bounded reachable state space up to a state in which the one-to-many agreement property is broken is too huge, which cannot be tackled by our parallel stratified testing technique. To alleviate the situation, we extend our parallel stratified testing in that we randomly select some states located at some depth from which we test states located at deep positions instead of exhaustively testing all states located at deep positions.

We first generate all states located at depth D_1 from each initial state for layer 1 (see Fig. 1). If D_1 is small enough, it is possible to do so. Given one initial state, there is one sub-state space in the first layer explored with JPF. From all states located at the bottom positions of layer 1 (or at layer 1), we specify a percentage of states to select states randomly for succeeding layer exploration instead of taking all states into account. In Fig. 1, the blue circles represent the selected states, while the yellow circles are ignored. For each one s among the randomly selected states at layer 1, we then generate all states located at depth D_2 (depth $D_1 + D_2$ from the initial state) (see Fig. 1) reachable from s. If D_2 is small enough and the number of the randomly selected states is small enough, it is possible to do so. There are as many sub-state spaces in layer 2 as the randomly selected states at layer 1.

The technique could be generalized such that the number of layers is $\mathrm{N} \geq 2$. We can do the same thing for the states at layer 2 as we did for the states at layer 1. We can randomly select states at layer 2 (or at depth $D_1 + D_2$ from the initial state) according to the percentage of the states generated at layer 2. Then, we can randomly select states at layer 3 (or at depth $D_1 + D_2 + D_3$ from the initial state) according to the percentage of the states generated at layer 3. It is worth mentioning that generating states for each sub-state space is independent of any other sub-state spaces in the same layer. This characteristic of the proposed technique makes it possible to generate states from concurrent programs at each layer in parallel.

V. Parallel stratified random testing for concurrent programs

From an initial state, we start verifying that programs enjoy desired properties in a stratified and randomized way such that it is possible to tackle multiple sub-state spaces in parallel. For each layer l, we specify a percentage based on which we randomly select states among those generated and located at the bottom positions of layer l (or at layer l). We do verification for each sub-state space in layer l at the same time when we generate states at layer l, and so do we for layer $l+1$. The technique to test concurrent programs in this way is called the parallel stratified random testing for concurrent programs.

Our tool supporting the parallel stratified random testing for concurrent programs has been implemented in Java. The tool architecture is depicted in Fig. 2, which is based on the master-worker model (or pattern). We use one master and four workers to conduct verification with our technique. The current number (4) of workers is tailored for the currently used computer, and can be increased and decreased depending on the computer used. We use Redis [7] and RabbitMQ [8] 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 hashes. It could hold its database entirely in memory. Hence, we use Redis as an effective cache to avoid duplicating states when generating states at each layer. We also use Redis to select states randomly given a percentage of states at each layer. Besides, we use Redis to store the status of workers running in our environment.
- RabbitMQ is used as a message broker. The RabbitMQ master maintains message queues to dispatch messages to RabbitMQ (RMQ) clients. Each worker consists of a RabbitMQ client and JPF.

In the beginning, we run a starter program to do several things for initialization. Firstly, we clean up everything from Redis cache as well as RabbitMQ master. Secondly, we send an initial state (as a message) to a message queue, where the initial state corresponds to the initial state specified in the specification concerned. States as messages are distributed to workers through the message queue. The initial state will be distributed to a worker that works on the first layer, generating all states located at (the bottom of) the first layer and do testing each state that occurs to all the states generated from the initial state as well. Let $\mathrm{depth}\text{-}D$ be a set of states located at depth D, where D is the depth from the initial state. If there are n states generated at depth D from the initial state and we specify $\delta\%$

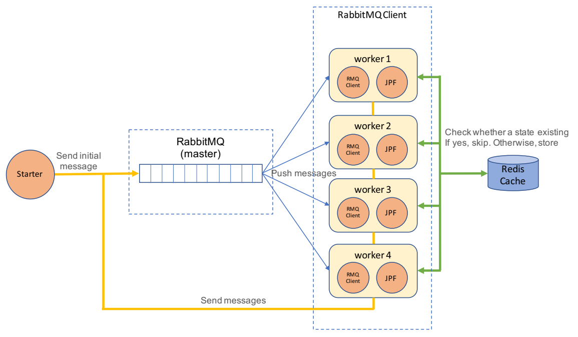

Fig. 2. The environment architecture

to randomly select those among the n states, depth-D is to contain approximately $0.01 \times \delta \times n$ states. We will describe why it contains approximately but not exactly $0.01 \times \delta \times n$ states. Because states are randomly selected, although the number of states in depth-D does not change drastically, depth-D can be drastically different from test to test. If there is one initial state, depth-0 contains the initial state. If the depth of layer 1 is D_1, the number of all states located at depth D_1 from the initial state is n_1 and the percentage of layer 1 is $\delta_1\%$, depth-D_1 contains approximately $0.01 \times \delta_1 \times n_1$ states located at depth D_1 from the initial state. In general, if $D_{(1,\ldots,l+1)}$ is the depth of layer $l+1$ from the initial state, n_{l+1} is the number of states located at $D_{(1,\ldots,l+1)}$ reachable from all states in depth-D_l and the percentage of layer $l+1$ is $\delta_{l+1}\%$, depth-$D_{(1,\ldots,l+1)}$ contains approximately $0.01 \times \delta_{l+1} \times n_{l+1}$ states located at depth $D_{(1,\ldots,l+1)}$ from the initial state. As depth-D, the number of states in depth-$D_{(1,\ldots,l+1)}$ does not change drastically but the states in depth-$D_{(1,\ldots,l+1)}$ can be drastically different from test to test. Besides, we store other initialization information to Redis, such as 0 as the current depth and 1 as the current layer in the beginning.

To make the environment working effectively, we use three message queues to store as well as dispatch states as messages to workers: QUEUE-1, QUEUE-2 and QUEUE-3. For example, we put all states generated and located at layer 1 into QUEUE-2. Let us suppose that the depth of layer 1 is D_1, the depth of layer 2 from the initial state is $D_{(1,2)}$ and the depth of layer 3 from the initial state is $D_{(1,2,3)}$. To work on states located at (the bottom positions of) layer 1, workers are supposed to fetch states as messages from QUEUE-2 and to check if the states are registered in depth-D_1. If so, workers generate the states located at depth $D_{(1,2)}$ reachable from the states fetched from QUEUE-2, putting the states generated into QUEUE-3 and depth-$D_{(1,2)}$ as well. Otherwise, workers just put away the states fetched from QUEUE-2. There may be the time when a worker (called the last worker for layer 2) fetches the last state from QUEUE-2 and the state is registered in depth-D_1. Then, the worker starts generating the states located at depth $D_{(1,2)}$ reachable from the state lastly fetched from QUEUE-2, when there may be some workers that have completely generated the states located at $D_{(1,2)}$ reachable from some states fetched from QUEUE-2. Such workers are called free workers for layer 2 because there are no more states left in QUEUE-2. Free workers for layer 2 do not need to wait until the last worker completes its current task. Thus, free workers starts working on states fetched from QUEUE-3. Note that such states are also registered in depth-$D_{(1,2)}$ because random selection of the states located at depth $D_{(1,2)}$ is not done. When the last worker for layer 2 completes its task, it does random selection of the states located at depth $D_{(1,2)}$. The last worker for layer 2 uses the percentage $\delta_2\%$ to delete some states from depth-$D_{(1,2)}$. Let n_2 be the number of states in depth-$D_{(1,2)}$ just before the random selection. Because some states may have been fetched from QUEUE-3 by some free workers for layer 2, the final result left in depth-$D_{(1,2)}$ by the random selection may contain a bit larger than $0.01 \times \delta_2 \times n_2$ states. This is the reason why depth-$D_{(1,2)}$ contains approximately but not exactly $0.01 \times \delta_2 \times n_2$ states.

The three message queues are used in turn. For example, QUEUE-1 is used to store the state located at depth 0 (namely the initial state), QUEUE-2 is used to store states located at depth D_1, QUEUE-3 is used to store states located at depth $D_{(1,2)}$, and then QUEUE-1 is used to store states located at depth $D_{(1,2,3)}$. Let us suppose that we only use two message queues QUEUE-1 & QUEUE-2 and worker 1 has just fetched the last message from QUEUE-2, which becomes empty at this moment. We also suppose that the other workers are free for some layer (say layer 2). If the free workers work on some states fetched from QUEUE-1 and generate & put some states into QUEUE-2, worker 1 will deal with the states newly put into QUEUE-2 incorrectly because worker 1 has not yet changed the queue from which messages are fetched. Thus, the free workers need to wait until worker 1 has fully completed its task, which is less efficient. If we use one more queue QUEUE-3, the free workers do not need to wait but can generate & put states into QUEUE-3. We do not need to use more than three message queues because after worker 1 has completed its task, QUEUE-2 will never become fully empty. When workers work on the final layer, they do not put any states into any message queues. If all workers are free for the final layer and there is no message in the three message queues, the verification is done.

Note that a message is a state. As soon as the RabbitMQ master has received a message, the RabbitMQ master stores the message in a designated message queue among those three message queues. By default, the RabbitMQ master will pop a message from the queue and then dispatch it to a worker in sequence. We suppose that all tasks have almost the same load, where each task is to generate all states located at some depth reachable from a given state and test each state that occurs from the given state to all the states located at the depth. Therefore, on average, every worker will get the same number of messages because we use the round-robin scheduling for distributing messages to workers. This makes the load labored by each worker well-balanced.

Regarding JPF workers, in the beginning, each worker needs to fetch the current status of the environment from Redis to initialize its own information, where the current status consists of information about whether some other workers are running and if so what message queue (among QUEUE-1, QUEUE-2 and QUEUE-3) the running workers fetch messages from. From the current status, the worker decides a message queue from which it fetches a message. If there are no workers running, QUEUE-1 is selected by default. Conversely, If there are some workers running to fetch messages from a message queue, say QUEUE-k, where k is 1, 2 or 3. We check if QUEUE-k has any messages. If so, QUEUE-k is selected. Otherwise, QUEUE-k', where $k' = (k \bmod 3) + 1$, is selected; after that the worker updates the current message queue being consumed to Redis, which makes it possible for the other workers to keep track on the up-to-date status of the environment. Note that every time a worker has changed the message queue from which messages are fetched, we need to update Redis about it. From the time on, workers listen to the message queue until messages arrive at the queue from which messages are dispatched to the workers or some existing messages in the queue are dispatched to the workers. Whenever a worker receives a message from the message queue, it will check whether or not the message exists in the randomly selected states at the current layer l stored in the set $\mathrm{depth}\text{-}D$ of states in Redis, where D is the depth of the layer l from the initial state. If no, we ignore the message and wait for a succeeding message. Otherwise, the worker internally starts a JPF instance with a configuration built from the message, generating all states located at some depth reachable from the message (a state) and testing each state that occurs from the message to all the states located at the depth. The configuration consists of values of observable components used in a program under verification. We then add our own listener class to JPF, which allows us to interact with JPF while executing. Note that all workers and JPF programs are running parallel and using the same Redis instance as a shared cache. For each layer exploration, we use a different set of states stored in Redis, namely $\mathrm{depth}\text{-}D$, where D is the depth of the layer from the initial state. After the layer exploration, based on a given percentage of states for the layer, we select states randomly among those in $\mathrm{depth}\text{-}D$ for the succeeding layer by deleting states randomly from the set $\mathrm{depth}\text{-}D$ of states in Redis.

Whenever JPF reaches the designated depth or finds that the current state has no more successor states, our listener class can listen to those events and do the following.

1) Looking inside the JPF heap of the current state to extract the values of observable components of the program; those values are then encapsulated into a $\mathrm{Configuration}$ object;
2) Checking if the state is in Redis cache; if yes, skipping what follows; otherwise, we ask the Redis to save the state to the set of states at the current depth and send the state to the next message queue among the three message queues; those messages in the next message queue are used to generate states in the next layer; note that, every time we have a new state that is jointly stored into Redis cache and send it to a message queue for consuming.

To control workers working smoothly with the three message queues, each worker is associated with a monitor thread to supervise the worker's activity and periodically check the status of the message queues by communicating with RabbitMQ Management HTTP API. Thereby, we can decide when a worker should change the message queue from which messages are fetched and which is the last worker left to switch to the next message queue.

VI. A CASE STUDY

Let us take the Needham-Schroeder Public-Key authentication protocol (NSPK) [6] as an example. NSPK can be described as three message exchanges:

$$\begin{aligned}&\text{Challenge: A} \to \text{B} : \{N_a, A\}_{K_b}\\&\text{Response: B} \to \text{A} : \{N_a, N_b\}_{K_a}\\&\text{Confirmation: A} \to \text{B} : \{N_b\}_{K_b}\end{aligned}$$

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 [9]. The corrected version is called NSLPK that can be described as follows:

$$\begin{aligned}&\text{Challenge: A} \to \text{B} : \{N_a, A\}_{K_b}\\&\text{Response: B} \to \text{A} : \{N_a, N_b, B\}_{K_a}\\&\text{Confirmation: A} \to \text{B} : \{N_b\}_{K_b}\end{aligned}$$

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 NSPK (but not NSLPK) in Maude. We use the following operators as the constructors of observable components:

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

where $\mathit{Soup}\{\mathit{Msg}\}$, $\mathit{Soup}\{\mathit{Rand}\}$, $\mathit{Soup}\{\mathit{Nonce}\}$ and $\mathit{Soup}\{\mathit{Prin}\}$ are the sorts for soups of messages, random numbers, nonces and principals, respectively. The nw observable component stores all messages sent by principals. The $rand$ 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 and three principals (two trustable ones and one intruder) and then the $rand$ 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 emp. The initial state denoted $init$ is as follows:

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

The actions of NSPK that exactly obey the protocol are specified in the following three rewrite rules:

```
rl [Challenge] : {(nw: NW) (nonces: Ns)
(rand: (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)) (rand: Rs) (prins: (P Q Ps))} .
```

```
rl [Response] : {(nw: (m1(P′,P,Q,c1(Q,N,P))
NW)) (rand: (R Rs)) (nonces: Ns) OCs}
=>
{(nw: (m2(Q,Q,P,c2(P,N,n(Q,P,R)))
m1(P′,P,Q,c1(Q,N,P)) NW)) (rand: 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′))
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′))
m1(P,P,Q,c1(Q,N,P)) NW))
(nonces: (if Q == intrdr then (N′ Ns)
else Ns fi)) OCs} .
```

The intruder can fake messages based on the nonces gleaned and the message in the network. The intruder's actions that fake messages are specified in the following six rewrite rules:

```
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′)) 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} .
```

Let S_{NSPK} refers to the specification of NSPK in Maude. A concurrent program P_{NSPK} is written in Java based on S_{NSPK}, where each thread of (two non-intruder and one intruder) principals performs the three actions $\mathrm{Challenge}$, $\mathrm{Response}$, and $\mathrm{Confirmation}$ and one thread of the intruder principal also performs the fake actions fake11, fake12, fake21, fake22, fake31, and fake32. When P_{NSPK} runs, it is necessary to select an initiator and a responder. If we select two principals as an initiator and a responder in a deterministic way, we may overlook several possible scenarios, which is not good from a verification point of view. We could use a pseudo-random number generator provided by Java to select principals. But, the pseudo-random generator cannot be controlled by JPF. To this end, JPF prepares $\mathrm{Verify.getInt}$ that generates pseudo-random numbers and can control it like the JPF Java VM. Given two integers min and max such that $\mathrm{min} < \mathrm{max}$, $\mathrm{Verify.getInt(min, max)}$ generates pseudo-random numbers between min and max. JPF takes all integers between min and max into account when it verifies a program in which $\mathrm{Verify.getInt(min, max)}$ is used.

To verify a concurrent program with JPF by using our technique proposed, we need to pass a message (state) directly to the Java program as a string argument. It is necessary to parse a string that denotes a state so that we can extract the value stored in each observable component. One state of NSPK protocol is as follows:

```
{nw: (
m1(p,p,intrdr,c1(intrdr,n(p,intrdr,r1),p))
m1(intrdr,p,q,c1(q,n(p,intrdr,r1),p))
m2(q,q,p,c2(p,n(p,intrdr,r1),n(q,p,r2)))
m2(intrdr,intrdr,p,c2(p,n(p,intrdr,r1),
n(q,p,r2)))
m3(p,p,intrdr,c3(intrdr,n(q,p,r2))))
rand: r1 r2
nonces: (n(p,intrdr,r1) n(q,p,r2))
prins: (p q intrdr)
rw_p: (Challenge Confirmation)
rw_q: (Confirmation)
rw_intrdr: emp}
```

We use Context-Free Grammar (CFG) with ANTLR library - a powerful parser generator for parsing these kinds of string messages [10]. Given grammar specified by an Extended Backus-Naur-Format (EBNF), ANTLR may generate a parser corresponding to the grammar. Basically, ANTLR does two phases. The first phase is to do a lexical analysis that breaks sentences into a series of tokens. The second phase is to do a syntax analysis. Given the series of tokens from the lexical analysis, the syntax analysis performs actual parsing, where the tokens are analyzed with the grammar for their structure such that a parse tree can be built as the output at the end. The following is the grammar of NSPK protocol used to generate a parser:

```
grammar Nspk;
start : '{' oc+ '}';
oc :
    'nw:' messagelist   # networkOC
    | 'rand:' randlist  # randOC
    | 'nonces:' noncelist   # noncesOC
    | 'prins:' prinslist    # prinsOC
    | rw rulelist #rwOC
    ;
rw : RW_P | RW_Q | RW_INTRDR ;
RW_P : 'rw_p:' ;
RW_Q : 'rw_q:' ;
RW_INTRDR : 'rw_intrdr:' ;
RULE : 'Challenge' | 'Response'
    | 'Confirmation' | 'Fake' ;
rulelist : RULE | RULE rulelist
    | '(' rulelist ')' | EMPTY ;
MESSAGENAME : 'm1' | 'm2' | 'm3' ;
message : MESSAGENAME '(' prin ',' prin ','
    prin ',' cipher ')' ;
messagelist : message | message messagelist |
    '(' messagelist ')' | EMPTY ;
prin : 'p' | 'q' | 'intrdr' ;
prinslist : prin | prin prinslist |
    '(' prinslist ')' | EMPTY ;
cipher : 'c1' '(' prin ',' nonce ',' prin ')'
| 'c2' '(' prin ',' nonce ',' nonce ')'
| 'c3' '(' prin ',' nonce ')' ;
nonce : 'n' '(' prin ',' prin ',' RAND ')' ;
noncelist : nonce | nonce noncelist |
    '(' noncelist ')' | EMPTY ;
RAND : 'r1' | 'r2' ;
randlist : RAND | RAND randlist
    | '(' randlist ')'| EMPTY ;
EMPTY : 'emp' ;
WS : [ \t\r\n]+ -> skip ;
```

Firstly, we need to generate an NSPK parser based on the given grammar by ANTLR. This parser is used to parse the NSPK messages that are NSPK states. Secondly, we need to write a Visitor class to extract all elements as well as corresponding values in the abstract parse tree. Basically, our Visitor class will subscribe to some events emitted from the parser while creating the abstract parse tree. Listening to these events, we can extract values correctly to initialize the values of the observable components in the program before starting.

For NSPK experiment, we suppose that there are two non-intruder and one intruder principals with two unique random values. The reachable state space of NSPK is too huge to exhaustively explore the entire reachable state space. Even though the depth bound is set to 200, the number of layers is 2 such that each layer depth is 100 and the multiple sub-state spaced obtained are tackled in parallel, the exhaustive exploration cannot be completed in three weeks. Therefore, we have conducted some preliminary experiments with parallel stratified random testing to demonstrate that the technique may mitigate the state space explosion to some extent. In the preliminary experiments conducted, the number of layers is three, each layer depth is 100 and the percentages of states generated for layer 1 and layer 2 are as follows: (1) 0.05% and 0.05%; (2) 0.05% and 0.1%; (3) 0.1% and 0.05%; and (4) 0.1% and 0.1%. Because layer 3 is the final layer, it is not necessary to specify the percentage of states generated. The experimental data are shown in Table I. The computer used for the experiments carried 2.9GHz micro-processor and 32GB main memory. Each experiment completes in 19h, while it does not complete to exhaustively test the reachable state space up to depth 200 in three weeks. The experimental data exhibit some mitigation of the state space explosion to some extent. In the experiments, however, we tried to detect a state in which the one-to-many agreement property is broken but did not find any such state, although we quickly found a state in which the nonce secrecy property is broken with parallel stratified (exhaustive but non-random) testing [5].

TABLE I
EXPERIMENTAL DATA FOR THE NSPK PROTOCOL

Total Depth	Layer 1(%)	Layer 2(%)	Worker	Time (d:h:m)
300	0.05	0.05	Worker 1	0:11:58
			Worker 2	0:12:03
			Worker 3	0:13:11
			Worker 4	0:11:43
300	0.05	0.1	Worker 1	0:07:56
			Worker 2	0:08:15
			Worker 3	0:06:42
			Worker 4	0:06:45
300	0.1	0.05	Worker 1	0:17:44
			Worker 2	0:18:51
			Worker 3	0:17:53
			Worker 4	0:17:51
300	0.1	0.1	Worker 1	0:16:29
			Worker 2	0:16:27
			Worker 3	0:15:00
			Worker 4	0:16:32

- Time – time taken to verify (sub-)state spaces with parallel stratified random testing.
- The number of layers used is three and each layer depths is 100.

VII. RELATED WORK

Model checking is a systematic way to verify concurrent programs by exhaustively searching all possible thread interleaving. However, it does not scale well on program size that often leads to the state space explosion. Random testing is an approach to avoiding the stuck with model checking. Therefore, several random testing techniques have been devised for testing concurrent programs. Most of them rely on randomized schedulers to explore the state space [11], [12], [13], [14], [15], [16].

RAPOS is an effective random testing algorithm for the partial order reduction technique [11]. At each state, there is a set of threads that could be selected to execute the next instruction. Rather than randomly choosing a thread, RAPOS calculates a set of threads such that their next instructions are independent. From the set of threads, they choose a random set of threads with a probability. Then the next instruction of threads in the random set of threads will be executed simultaneously without care about the order of instructions. PCT (Probabilistic Concurrent Testing) [14] is another randomized algorithm for concurrency testing based on a priority scheduler where the highest priority for a thread at each scheduling step will be selected. In the beginning, PCT assigns priority values randomly to threads and creates random places (or change points) where the scheduler could change the priorities of threads during execution. It has been proved that PCT can detect bugs with an efficient probability. By combination of random testing with parallelization, parallel randomized state-space search [12] has been proposed. They used a revised version of the depth-first search for randomly selecting the next state among all successor states of a current state. A parallel version was also developed to check the same state space with multiple machines independently, making it possible to quickly find a bug. Along with the use of the depth-first search, the DFS-RB algorithm has been introduced [15]. Basically, they use the depth-first search as usage with early backtracking. To decide when and how many steps are backtracked, DRS-RB uses some values of parameters to decide it at a state during searching. Recently, they have adaptive randomized scheduling (ARS) [16], which uses memory access patterns (with 17 common patterns) to calculate the distance between two traces at each scheduling point, where a trace is a path from a source state to a target state in a search tree. At each step, ARS maintains only N traces whose distances are largest. If there is the same distance between traces. ARS randomly selects N traces from all traces. After some steps, the last N traces are most likely to lead to a bug. So ARS continuously does verification based on that N traces.

As described above, any existing randomized testing techniques are different from our parallel stratified random testing for concurrent programs in some aspects. They mostly use randomize schedulers with or without a criterion for selecting a thread to execute a next instruction. Our approach uses a percentage to select states randomly at each layer for the succeeding layer exploration. We also develop a parallel version to improve the effectiveness of sub-state spaces exploration.

VIII. Conclusion

We have proposed parallel stratified random testing for concurrent programs. For a Java implementation of the NSPK authentication protocol, even though we use 2 layers and each layer depth is 100, it did not complete in three weeks to exhaustively test the concurrent program. On the other hand, when we used 3 layers, each layer depth was 100 and the percentages for layer 1 & layer 2 were 0.05% or 0.1% & 0.05% or 0.1%, respectively, it completed in 19h to test the concurrent program. Although we still did not find a state in which the one-to-many agreement property is broken, we made some progress toward making it possible to detect such a state.

One piece of our future work is to detect a state in which the one-to-many agreement property is broken by increasing the percentages for layer 1 and/or layer 2. Even though the percentages for layer 1 and/or layer 2 are increased, we might not find such a state. If so, we need to use some other criterion on which states are selected in addition to random state selection and/or we may need to use randomized schedulers as the existing related techniques do.

References

[1] C. M. Do and K. Ogata, "Specification-based testing with simulation relations," in *31st SEKE*, 2019, pp. 107–146.

[2] ——, "A divide & conquer approach to testing concurrent Java programs with JPF and Maude," in *9th SOFL+MSVL*, ser. LNCS, vol. 12028, 2019, pp. 42–58.

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

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All About Maude*, ser. LNCS. Springer, 2007, vol. 4350.

[5] C. M. Do and K. Ogata, "A divide & conquer approach to testing concurrent programs with JPF," Submitted for publication, 2020.

[6] R. M. Needham and M. D. Schroeder, "Using encryption for authentication in large networks of computers," *Comm. ACM*, vol. 21, no. 12, pp. 993–999, 1978.

[7] Open source, "Redis," https://redis.io/, 2009, [Online; accessed 05-August-2019].

[8] ——, "Rabbitmq," https://www.rabbitmq.com/, 2007, [Online; accessed 05-August-2019].

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

[10] Open source, "Antlr," https://www.antlr.org/, 2014, [Online; accessed 05-March-2020].

[11] K. Sen, "Effective random testing of concurrent programs," in *22nd ASE*, 2007, p. 323332.

[12] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare, "Parallel randomized state-space search," in *29th ICSE*, 2007, pp. 3–12.

[13] K. Sen, "Race directed random testing of concurrent programs," in *PLDI 2008*, 2008, pp. 11–21.

[14] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," in *15th ASPLOS*, 2010, pp. 167–178.

[15] P. Parízek and O. Lhoták, "Fast detection of concurrency errors by state space traversal with randomization and early backtracking," *Int. J. Softw. Tools Technol. Transf.*, vol. 21, no. 4, pp. 365–400, 2019.

[16] Z. Wang, D. Zhang, S. Liu, J. Sun, and Y. Zhao, "Adaptive randomized scheduling for concurrency bug detection," in *24th ICECCS*, 2019, pp. 124–133.