I. CLOUDSYNC IMPLEMENTATION IN JAVA

The class diagram depicted in Fig. 1 shows how to implement the protocol in Java, where $Cloud$ and PC classes correspond to the $Cloud$ and the PC in the specification, respectively. The PC class extends $Thread$ class to construct a multi-threaded program. The status of the $Cloud$ is represented by an enum $LabelC$ type, which is in the form of either $idlec$ or $busy$, while the status of a PC is represented by an enum $LabelP$ type, which is in the form of either $idlep$, $gotval$ or $updated$. Main functionalities are implemented inside the PC class that has the methods $getval$, $updated$ and $gotoidle$ corresponding to $gotval$, $update$ and $gotoidle$ rewrite rules described above, respectively.

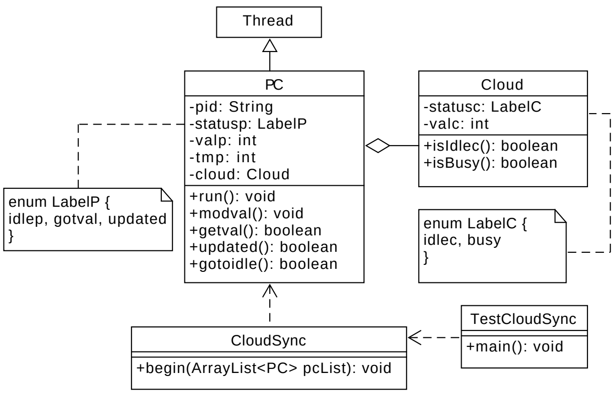

FIGURE 1: CloudSync class diagram

The $main$ method in the $TestCloudSync$ class creates and initializes one $Cloud$ instance and some PC instances, when you can decide the number of PC instances. The $Cloud$ instance is associated with a list of PC instances. The list of PC instances is passed to the $begin$ method of the $CloudSync$ class. From the $CloudSync$ class side, it starts all PC instances running as threads. The run method in each PC is invoked and mainly consists of a while $true$ statement inside which $getval$, $updated$, and $gotoidle$ methods are called in order. The methods synchronize with the $Cloud$ instance to make sure that there is no race condition occurring when PC instances run in parallel, working on the operations of the $Cloud$ instance.

Let us recall what we need to do to generate state sequences from the program [1] implemented in Java as follows. Whenever a worker receives a message from the message broker, it internally starts a JPF instance. Given a Java program as an input to the JPF instance, it works on generating state sequences. To this end, we need to pass a message to the Java program as a string argument to tell the Java program the initial values of the observable components. On receipt of such a string message, the Java program needs to parse the string to recognize the initial value of each observable component in the beginning. A Revised CloudSync message (that represents a state) may look like as follows:

```
{(cloud: < idlec,2 >) (pc[p1]: < idlep,1,0 >)
 (pc[p2]: < idlep,2,0 >)
 (pc[p3]: < idlep,3,0 >)}
```

Each of (cloud: ...), (pc[p1]: ...), (pc[p2]: ...) and (pc[p3]: ...) is a name/value pair. $p1$, $p2$ and $p3$ represent three PCs that are used in the program. Although it suffices to use regular expressions to parse the message, we have encountered complicated messages while tackling the NSLPK case study that is reported in the next sub-section. To make it possible to parse complicated messages, we decided to parse messages by using Context-Free Grammar (CFG) with ANTLR library - a powerful parser generator [2]. Given grammar that is specified in an Extended Backus-Naur Form (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 a string 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 Revised CloudSync case study used to generate a parser:

```
grammar CloudSync;
start : CURLY_BRACKETS_OPEN oc+
    CURLY_BRACKETS_CLOSE ;
oc : PARENTHESES_OPEN oc PARENTHESES_CLOSE |
    cloud | pc ;
cloud : CLOUD_ID COLON LESS_THAN LABELC COMMA
    INTEGER_NUMBER GREATER_THAN ;
CLOUD_ID : ’cloud’ ;
pc : PC_ID SQUARE_BRACKETS_OPEN PID
    SQUARE_BRACKETS_CLOSE COLON
    LESS_THAN LABELP COMMA INTEGER_NUMBER
    COMMA INTEGER_NUMBER GREATER_THAN ;
PC_ID : ’pc’ ;
LABELC : ’idlec’ | ’busy’ ;
LABELP : ’idlep’ | ’gotval’ | ’updated’ ;
PID : ’p’ INTEGER_NUMBER ;
LESS_THAN : ’<’ ;
GREATER_THAN: ’>’ ;
PARENTHESES_OPEN : ’(’ ;
PARENTHESES_CLOSE : ’)’ ;
CURLY_BRACKETS_OPEN : ’{’ ;
CURLY_BRACKETS_CLOSE : ’}’ ;
SQUARE_BRACKETS_OPEN : ’[’ ;
SQUARE_BRACKETS_CLOSE : ’]’ ;
INTEGER_NUMBER : DIGIT+ ;
fragment
DIGIT : (’0’..’9’) ;
COLON : ’:’ ;
COMMA : ’,’ ;
EMPTY : ’emp’ ;
WS : [ \t\r\n]+ -> skip ;
```

Given the grammar and the input message above, ANTLR can generate an abstract parse tree. Firstly, we need to generate a Revised CloudSync parser based on the given grammar by using ANTLR. This parser is used to parse Revised CloudSync messages. Secondly, we write a $Visitor$ class to extract all observable components (including their values stored) in the abstract parse tree. Basically, our $Visitor$ class subscribes to some events emitted from the parser while visiting at the abstract parse tree. Listening to these events, we can get values and then initialize observable components in the Java program before starting generating state sequences.

II. NSLPK IMPLEMENTATION IN JAVA

The class diagram depicted in Fig. 4 shows how to implement the protocol in Java where there are two non-intruder principals, one intruder, and two random numbers. Based on the specification described above, we define $Rand$, $Nonce$, $Message$, and $Cipher$ classes shown in Fig. 2. $Rand$ class has attribute id whose type is $String$ and that is used to represent either $r1$ or $r2$. The $Nonce$ class contains a $Rand$ object r and two Principal objects $p1$ and $p2$ where $p1$ is the generator and $p2$ is the principal to whom $p1$ wants to make a session. To construct a $Nonce$ object, one $Rand$ object and two $Principal$ objects are used. $Cipher1$, $Cipher2$, and $Cipher3$ classes are prepared to represent three kinds of ciphertexts used in $Challenge$, $Response$, and $Confirmation$ messages, respectively. We create a $Cipher$ interface that is implemented by three classes $Cipher1$, $Cipher2$, and $Cipher3$. The attributes of $Cipher1$, $Cipher2$ and $Cipher3$ classes are the same as those described in the specification. To construct $Challenge$, $Response$, and $Confirmation$ messages, we create a generic *Message<E:Cipher>* class in which E can either be $Cipher1$, $Cipher2$, or $Cipher3$. *Message<E:Cipher>* has attribute $cipher$ whose type is E and attribute $name$ whose type is $String$, where $name$ is used to express the kind of messages, namely $Challenge$ (or $m1$), $Response$ (or $m2$), and $Confirm$ (or $m3$) messages.

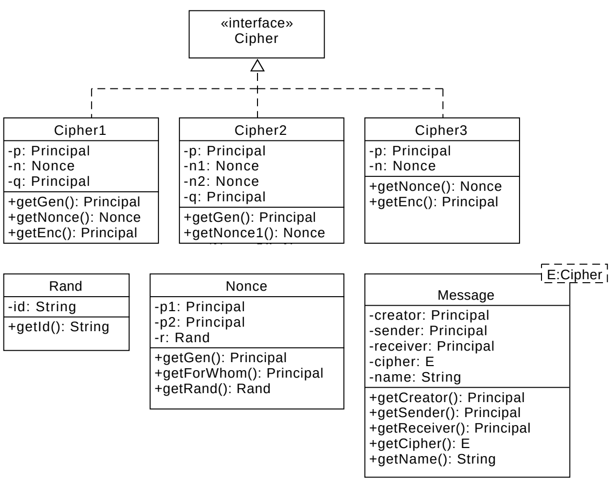

FIGURE 2: Class diagram for Rand, Nonce, Cipher and Message

Principals are implemented as $Principal$ class that extends $Thread$ class. Among attributes in $Principal$ are as follows: (1) id whose type is $String$ and that expresses a principal ID, such as p, q and $intrdr$, (2) nw whose type is *Network<Message<Cipher>>* and that contains all messages in the network, (3) $rands$ whose type is *Multiset<Rand>* and that keeps all random numbers available, (4) $prins$ whose type is *Multiset<Principal>* and that keeps all principals participating in the protocol, and (5) $nonces$ whose type is *Multiset<Nonce>* and that keeps all nonces gleaned by the intruder. nw, $rands$, $prins$, and $nonces$ are associated with all $Principal$ objects, although $nonces$ is only used by the intruder.

The $Principal$ class defines $challenge$, $response$, and $confirmation$ methods shown in Fig. 4 that correspond to the $Challenge$, $Response$, and $Confirmation$ transition rules, respectively. Besides, we have $fake$ and $do\_intruder$ methods that are specialized to the intruder. To implement the intruder, $Intruder$ class is created that extends $Principal$ class. In addition, $Intruder$ class has some more methods such as $fake11$, $fake12$, $fake21$, $fake22$, $fake31$, and $fake32$ methods corresponding to those fake transition rules in the specification. The $Intruder$ class needs to overwrite $fake$ and $do\_intruder$ methods. In $fake$ method, each fake method mentioned above is called one by one in order. $do\_intruder$ method is regarded as a trigger method, namely that if a $Principal$ object calling this method is an $Intruder$ object, then $do\_intruder$ method adds gleaned nonces to the list of nonces, namely $nonces$.

In theory, principals run in distributed mode and we do not control which principal should make a session to which principal at all. Hence, when principals run, everything is randomly chosen. For example, two principals and one random number are randomly chosen from $prins$ and $rands$, respectively, to make a $Challenge$ message. To accomplish it, we define a $Controller$ class (see Fig. 4). Given a list of elements, the $Controller$ will pick up one element randomly in the list by $getOne$ method. After that, we may get the next element randomly, excluding the one chosen most recently, in the list by $getNext$ method. Our purpose is to pick up an element E in a list of elements E randomly at runtime. However, we need to control elements chosen randomly when generating state sequences from the program with JPF. Hence, the pseudo-random number generator supported by Java can not be used in this case. We use $Verify.getInt(min, max)$ method supported by JPF. Given min and max values, the method will randomly return a value between min and max. The key point is that when JPF runs to check the program and meets $Verify.getInt(min, max)$ method, JPF evaluates all possible values between min and max to the program.

To implement the transition rules in Java, we define a $RewriteRule$ interface shown in Fig. 3 that has $execute$ method with a $Principal$ as a parameter. $Challenge$, $Confirmation$, $Response$, and $Fake$ classes implement $RewriteRule$ interface (see Fig. 3). Inside the $execute$ method of each class, $challenge$, $confirmation$, $response$, and $fake$ methods are called from given $Principal$ p in $Challenge$, $Confirmation$, $Response$, and $Fake$ classes, respectively.

There is an attribute called $rwController$ in $Principal$ class (see Fig. 4). It maintains a list of rewrite rule objects

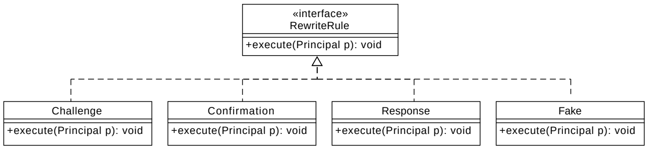

FIGURE 3: Rewrite rule implementation in Java

from which a rewrite rule is randomly chosen by a principal. Given a principal as a parameter, the chosen rewrite rule is executed for the principal. For a principal p to carry out the implementation of a rewrite rule, it is necessary to choose a principal q to which p wants to make a session. If p is not the intruder, q should be different from p. Otherwise, the intruder may need to choose two principals to fake messages. When so, one of the two principals may be the intruder.

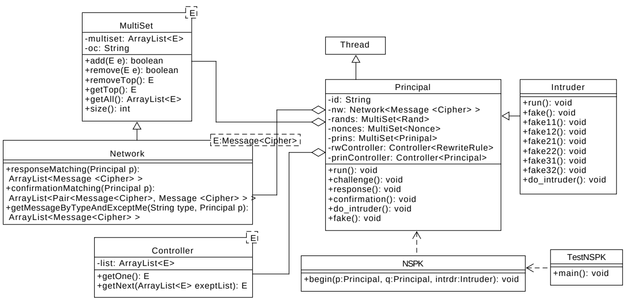

FIGURE 4: NSLPK class diagram

In the $main$ method of $TestNSLPK$ class, we create two non-intruder principal objects, one intruder object, and two random numbers to make nonces, initializing the attributes nw, $rands$, and $nonces$. Given the three principal objects to $begin$ method of $NSLPK$ class, it starts the principals running as threads. The run method in each principal object is invoked to initialize some attributes, such as $prinController$, and performs as follows:

```
while(true) {
 RewriteRule rr = this.rwController.getOne();
 rr.execute(this);
}
```

Principals use $rwController$ to pick up randomly a rewrite rule among $Challenge$, $Response$, $Confirmation$, and $Fake$ rewrite rules. Note that only the intruder principal has $Fake$ rewrite rule in the list. After picking up one, the rewrite rule calls the $execute$ method to execute the corresponding method by giving the current principal object as a parameter. If $Challenge$ is the rewrite rule picked up, the $execute$ method will call the $challenge$ method with the given principal object. The following is the $challenge$ method in $Principal$ class.

```
public void challenge() {
 synchronized (rands) {
  boolean is_empty = rands.isEmpty();
  if (!is_empty) {
   Principal p = this; // sender
   // randomly select receiver
   Principal q = this.prinController.getOne();

   Rand r = rands.removeTop();
   Nonce n = new Nonce(p, q, r);
   Cipher1 c1 = new Cipher1(q, n, p);
   Message<Cipher> m1 = new
   Message<Cipher>(Constants.m1, p, p, q, c1);
   nw.add(m1)
   q.do_intruder(n);
   // once Sender use 'challenge', should not
   // use 'challenge' and 'response' anymore
   this.rwController.remove(new Challenge());
   this.rwController.remove(new Response());

   // no other principals should use 'challenge'
   ArrayList<Principal> prins =
   this.getPrins().getAll();
   for (int i = 0; i < prins.size(); i ++) {
    prins.get(i).getRwController().remove(
    new Challenge());
   }
  }
 }
}
```

The principal object checks if $rands$ is not empty. If so, it can make a $Challenge$ message. Let r be the random number chosen. r is deleted from $rands$. Let p and q refer to the current principal and the one to which the message is sent. q is randomly selected with $getOne$ method from $prinController$ object. A nonce (referred to as n) is made from p, q, and r, a $Cipher1$ object is made and then a *Message<Cipher>* object (referred to as $m1$) is made. After that, $m1$ is added to the network and then the *do_intruder* method is called with n as its parameter. If p is the intruder, n is gleaned by the intruder, being added to $nonces$. There are only two random numbers available because otherwise the reachable state space becomes huge. Thus, if p sends a $Challenge$ message to some principals again, nothing interesting will happen. Hence, we remove the $Challenge$ rewrite rule from the p's $rwController$. Because of the same reason, we also remove the $Response$ rewrite rule from it and the $Challenge$ rewrite rule from the other principals' $rwControllers$. If p is the intruder, $Fake$ rewrite rule may be picked up. If that is the case, the $fake$ method in $Intruder$ class is invoked, where $fake11$, $fake12$, $fake21$, $fake22$, $fake31$, and $fake3$ methods are executed in order.

The $fake11$ method randomly selects a nonce (referred to as n) from $nonces$ if it is not empty. To this end, a *Controller<Nonce>* object $nonceController$ is made. Two principals p and q are randomly selected with $prinController$. n, p, and q are used to make a $Cipher1$ object and then a $Challenge$ message (referred to as $m1\_fake$) is faked. Then, $m1\_fake$ is added to the network referred to as nw.

```
if (!this.nonces.isEmpty()) {
  Controller<Nonce> nonceController = new
  Controller<Nonce>(this.nonces.getAll());
  // randomly select one Nonce
  Nonce n = nonceController.getOne();

  // sender
  Principal p = this.prinController.getOne();
```

```
  // receiver
  Principal q = this.prinController.getNext(
  new ArrayList<Principal>(Arrays.asList(p)));

  Cipher1 c1 = new Cipher1(q, n, p);
  Message<Cipher> m1_fake = new
  Message<Cipher>(Constants.m1,this,p,q,c1);
  nw.add(m1_fake)
}
```

Once NSLPK has been implemented in Java, we can start generating state sequences from the program. A state in such state sequences may look like 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))
m1(intrdr,p,q,c1(intrdr,n(p,intrdr,r1),p))
m3(intrdr,p,q,c3(q,n(p,intrdr,r1)))
m2(q,q,p,c2(p,n(p,intrdr,r1),n(q,p,r2),q))
m1(intrdr,q,p,c1(intrdr,n(p,intrdr,r1),p))
m2(intrdr,p,intrdr,c2(p,n(p,intrdr,r1),
   n(q,p,r2),q))
m2(intrdr,q,p,c2(p,n(p,intrdr,r1),
   n(q,p,r2),q)))
rands: emp
nonces: (n(p,intrdr,r1))
prins: (p q intrdr)
rw_p: (Confirmation)
rw_q: (Confirmation)
rw_intrdr: (Confirmation Fake)}
```

Such states are complicated enough mainly because the network contains various kinds of messages. As described, therefore, we use CFG with ANTLR library to parse such messages. The following is the grammar used in the NSLPK case study:

```
grammar Nslpk;
start : '{' oc+ '}';
oc :
    'nw:' messagelist   # networkOC
    | 'rands:' randlist  # randsOC
    | '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 ','
    prin ')'
    | '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 ;
```

Some annotations are used in the grammar, such as $\#networkOC$, $\#randsOC$, $\#noncesOC$, $\#prinsOC$, and $\#rwOC$. These annotations ask ANTLR to generate some extra methods in a listener class when generating the parser. Subscribing to these extra events and some other events in the listener class allows us to extract all messages in the network `nw`, all nonce values in `nonces`, all principals in `prins`, all random numbers in `rands`, and the rewrite rule list of each principal.

III. ORIGINAL CLOUDSYNC IMPLEMENTATION IN JAVA

We use the class diagram depicted in Fig. 1 to show how to implement the CloudSync in Java. Looking at the PC class (see Fig. 1), the $modval$ method is defined to implement the behavior of the $modval$ rewrite rule. You may imagine that the implementation of CloudSync is the same as Revised CloudSync. However, we need to revise the $getval$ method in which the $modval$ method is invoked. Thereby, the $modval$ method is combined to the $getval$ method so that the $valp$ of the PC always gets fresh before exchanging messages with the $Cloud$. The other parts of the implementation of CloudSync are the same as that of Revised CloudSync.

REFERENCES

[1] C. Minh Do and K. Ogata. A divide & conquer approach to testing concurrent java programs with jpf and maude. In H. Miao, C. Tian, S. Liu, and Z. Duan, editors, Structured Object-Oriented Formal Language and Method, pages 42–58, Cham, 2020. Springer International Publishing.

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