Electronic Notes in Theoretical Computer Science 71 (2003)
URL: `http://www.elsevier.nl/locate/entcs/volume71.html` 17 pages

A Secret-Sharing Protocol Modelled in Maude

Dilia E. Rodríguez [1,2]

Information Directorate, Air Force Research Lab., Rome, NY 13441-4505, USA.

Abstract

Meta-objects are used to factor the specification of the first asynchronous, proactive, secret-sharing protocol into secret-sharing, replication and encryption layers. A search strategy is described that exploits the structure of the specification to reduce the search space.

Key words: Protocol specification, meta-objects

1 Introduction

The demands on software systems are ever increasing. They should be distributed, reliable, adaptable, secure and more. Designing and reasoning about such systems is very difficult, and much can be at stake. To help manage this complexity Maude [1] brings into the formal arena object-orientation, inheritance, richly parameterized modules and user-definable syntax. To provide further means of modularizing, adapting and reusing formal specifications, meta-objects have been given a rewriting semantics [2]. Case studies can be useful in gaining experience in the judicious combination of these means. The work described here is an attempt in that direction.

COCA [7] is a fault-tolerant and secure on-line certification authority that has very weak assumptions, thus reducing its vulnerability. To maintain the private key used for signing certificates and responses the first asynchronous proactive secret-sharing protocol (APSS) was developed, which is the protocol described here.

In [6] it is described by a sequence of versions with increasingly weaker assumptions. The development of the specification and this presentation follow in those steps. Section 2 introduces secret sharing. The notions of objects and meta-objects in Maude, which are used to represent the system, are briefly described in Section 3. The first version of the protocol is presented in Section 4, introducing the basic structure of the specification. Sections 5 and 6

[1] Supported by the Air Force Office of Scientific Research.
[2] Email: `Dilia.Rodriguez@rl.af.mil`

describe modifications that obtain the next versions of the protocol. The correctness criteria for the protocol are defined in Section 7. Section 8 describes a search strategy that exploits the layered structure of the specification. A brief conclusion follows.

2 Secret Sharing

Secret sharing resolves the tension between the availability and the secrecy of a secret by distributing a secret to multiple servers in such a way that it is disclosed only if some of these servers are compromised. One particular scheme is $(n, t+1)$ secret sharing, defined by the operations *split* and *reconstruct*. Split generates from a secret a random set of n *shares*, called a *sharing*, while reconstruct obtains the secret from any subset of a sharing with more than threshold t shares.

A secret sharing is called standard if the shares are single values. In [6] Zhou describes the construction of an $(n, t+1)$ combinatorial secret sharing from an (l, l) standard secret sharing, with $l = \binom{n}{t}$. Sets $P_1, \ldots, P_l$ of t servers represent the worst-case failure scenarios. Given a sharing $s_1, \ldots, s_l$ of a secret, each share s_i is associated with failure scenario P_i. The share for server p of the $(n, t+1)$ secret sharing of the same secret, called the *share set* of p and denoted S_p, is defined by $s_i \in S_p \Leftrightarrow p \notin P_i$. More abstractly, to every server p there corresponds an *index set* I_p such that $i \in I_p \Leftrightarrow p \notin P_i$.

The secret-sharing protocol presented in [6] employs combinatorial $(n, t+1)$ secret sharing to maintain a secret. The next section presents very briefly the notions of object and meta-object in Maude, which will be used to represent servers and to separate different aspects of the protocol.

3 Objects and MetaObjects

A distributed object-oriented system is represented in Maude [4] as a term of sort `Configuration`, which has subsorts `Object` and `Msg`. Configurations are built by a multiset union operator, with a state of a distributed system described by a term:

$$M_1 \ldots M_m \quad < O_1 : C_1 | atts_1 > \ldots < O_n : C_n | atts_n > .$$

The Ms are messages, and the other terms give the states of objects named O_i of class C_i, with the values of their attributes given in $atts_i$, for $i = 1, \ldots, n$.

The dynamics of the system is specified by rewrite rules, whose most general form is

$$\begin{aligned} r: \quad & M_1 \dots M_m \quad < O_1 : C_1 | atts_1 > \dots < O_n : C_n | atts_n > \\ \rightarrow \quad & < O_{i_1} : C'_{i_1} | atts'_{i_1} > \dots < O_{i_n} : C'_{i_n} | atts'_{i_n} > \\ & < Q_1 : D_1 | atts''_1 > \dots < Q_p : D_p | atts''_p > \quad M'_1 \dots M'_q \\ \text{if } & Cond \qquad . \end{aligned}$$

This rule, labelled by r, states that if the condition $Cond$ holds the messages M are consumed; some of the original objects O persist with new states and possibly new classes C', and new objects Q and messages M' emerge. Maude supports the convention that a rule need not mention attributes of an object that are irrelevant for that rule.

To represent more complex entities there is the notion of meta-object, which is given a rewriting semantics in [2]. Different communication services can be represented by different meta-objects, and can be composed by stacking the corresponding meta-objects, perhaps subject to some interface constraints.

These stackable objects must be defined in a subclass of `MetaObject`, a class that allows messages to move down, and configuration requests, which are multisets of requests for message transmittal or object creation, to move up. `MetaObject` has five attributes: `base`, which identifies the meta-object at the bottom of the stack; and four lists of data that allow communication with neighboring meta-objects and the environment. Messages from above or the outside are imported into `in`; `down` holds the messages to be exported; `up` holds configuration requests imported from below; `out`, those being exported. In the specification presented here, `MetaObject` departs from the above description in that the importing lists are replaced by multisets.

A `MetaStack` is a list of `MetaObjects`, all of which have the same `base`, and whose last meta-object is the one identified as the base. A term of sort `MetaTower` has the form $\{MS\}$, where MS is a meta-stack. `MetaStack` and `MetaTower` are subsorts of `Configuration`.

The next section presents the first model of APSS. A configuration of meta-towers, objects and messages describes the state of the system. .

4 APSS with a Single Coordinator

This first protocol assumes a benign environment. It is an asynchronous protocol that has no bound on either the server speed, the message delay, or the local clock drift. Its goal is to proactively replace one sharing of a secret, collectively held by n servers, with another sharing of the same secret. A sharing is produced during one run or instance of the protocol and destroyed during the next run. To know the secret an adversary must obtain enough shares of a sharing to reconstruct it. Thus, the window of vulnerability is the duration

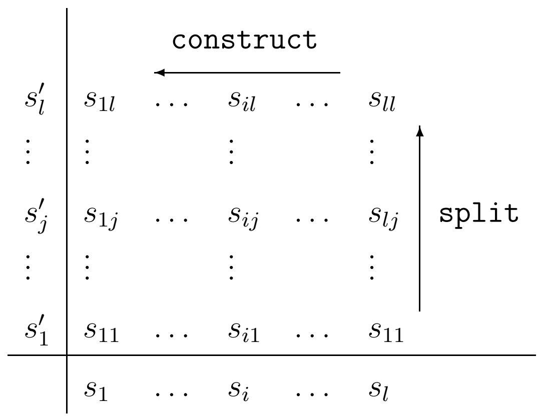
construct
split
s'_l
s'_j
s'_1
s_{1l} ... s_{il} ... s_{ll}
s_{1j} ... s_{ij} ... s_{lj}
s_{11} ... s_{i1} ... s_{11}
s_1 ... s_i ... s_l

Fig. 1. Share Refreshing

of two consecutive runs. The protocol assumes that during this time no more than t servers are ever compromised. It further requires $n \geq 3t + 1$. This first version assumes adversaries are passive. A compromised server can experience only crash failures, disclosing locally stored information to adversaries. Links are assumed to be reliable, delivering every message to its intended recipient; an adversary may only eavesdrop on links.

4.1 Share Refreshing

An $(n, t+1)$ combinatorial secret sharing is built on top of an (l, l) standard secret sharing, with $l = \binom{n}{t}$. Each server holds a standard share for each index in its index set. To replace a sharing $s_1, \ldots, s_l$, each of the shares is split, and the resulting subshares are reconstructed, as shown in Figure 1, to obtain $s'_1, \ldots, s'_l$.

The protocol has each uncompromised server split its shares. As each standard share is held by multiple servers, and the split operation is nondeterministic, it requires a coordinator to select the subsharings for the next sharing. This version assumes the coordinator is immune to attacks.

The syntax of the basic secret-sharing operations is given by

```
op split : MachineInt Label SplitId Secret -> NeSetOfShares .
op construct : MachineInt Label NeListOfShares -> Share? .
```

Both require the length of a sharing, and both label their result. Since `split` is nondeterministic split ids distinguish different applications. The protocol does not ever reconstruct the maintained secret. `construct` obtains a `Share`, which is a `Secret`, from a list of shares which must have the length of a sharing; otherwise, it returns `errorShare`.

Indexed sets of shares represent share sets of servers, and the collection of subshares they get. The basic operations are overloaded to operate on them.

```
op split : MachineInt Label MachineInt Oid NeISetOfShares
        -> SubsharesLabels .
```

Each of the shares s_i in the indexed set having the given label L is split into a (sub)sharing of the length given by the first argument and labelled by $L \circ i \circ O$, where O is the fourth argument, the name of the server performing the split. Each resulting sharing is further identified by a split id composed of the name of the server and a split count, which starts at the third argument.

```
op construct : MachineInt Label NeListOfLabels NeISetOfShares
            -> NeISetOfShares .
```

A new share set is produced by `construct`, whose precondition requires, for each index in the given indexed set, shares with each of the labels in the list, which must have the length given in the first argument. The resulting shares are labelled with the given label.

Operations to collect shares and delete all shares or all but those with some given label are introduced.

```
op _+>_ : NeISetOfShares NeISetOfShares -> NeISetOfShares .
op _>- :  NeISetOfShares -> NeISetOfShares .
op _>-`[_`] : NeISetOfShares Label -> NeISetOfShares .
```

4.2 APSS and MetaObjects

The asynchronous, proactive secret-sharing protocol is about splitting and constructing shares, but the replication and encryption required complicate it much. This specification factors these aspects into different meta-objects.

4.2.1 Secret-sharing Layer

The base layer captures the properly secret-sharing part of the protocol. The servers are holders of a sharing of a secret, and the coordinator guides them as they replace it. Messages are kept to a minimum and simple.

This section describes an asynchronous object rewrite theory, one whose rules have one object and at most one message on the left-hand side, which can then be transformed into a meta-object theory as described in [2].

Known to each participant or principal of a run is the length of a sharing and the version of the current one. Its status may be either `up` or `down`.

```
class Principal |               sharing-length : MachineInt,
    status : Status,            current-version-number : MachineInt .

class Server |
    split-count : MachineInt,   runs : SetOfRuns,
    shares : ISetOfShares,      subshares : ISetOfShares .
subclass Server < Principal .
```

```
class Coordinator |  run-in-progress : Bool,
    chose : Bool,    propagated-subsharings : PropagatedSubsharings .
subclass Coordinator < Principal .
```

It is convenient to distinguish between the identity and the behavior of a principal. Thus, only objects of the following classes participate in the protocol.

```
class ActiveServer .       subclass ActiveServer < Server .
class ActiveCoordinator .  subclass ActiveCoordinator < Coordinator .
```

The coordinator starts a run by instructing all servers to split their shares.

```
rl [init-run] :
    [ < C : ActiveCoordinator |
            status : up,        run-in-progress : false,
            chose : false,      current-version-number : V > Conf ]
 => [ < C : ActiveCoordinator | run-in-progress : true >
      ( all <| C |< init(label(V)) )                           Conf ] .
```

The syntax for the communications exchanged is given by

```
sorts MsgPkt ConfReq SMsg Message .
sorts Id AId .                         subsorts Oid AId < Id .
subsort MsgPkt < ConfReq < Msg .       subsort SMsg < Msg .
op _|<_ : Id Message -> SMsg .         op _<|_ : Id Msg -> MsgPkt .
op __ : ConfReq ConfReq -> ConfReq [assoc comm id: none] .
```

To abstract the replication from the secret-sharing part of the protocol terms of sort `AId` are used to represent one or more concrete objects, which are identified by terms of sort `Oid`. Above, `all` is used to represent all the servers. Abstract messages will be translated into messages to concrete objects by the replication meta-object.

As each server receives the instruction from the coordinator, it splits its shares with the specified label, and sends the subshares to the holders that should have them (see Figure 1) and the labels of the sharings produced to the coordinator.

```
crl [contribute] :
    [ < Srv : ActiveServer |
             status : up,                 runs : Rs,
             current-version-number : V,  split-count : Cnt,
             sharing-length : Len,        shares : Ss >
      ( Srv <| C |< init(L) )                                   Conf ]
 => [ < Srv : ActiveServer |              runs : Rs U (L, only-old),
             split-count : Cnt
                    + size(labels(split(Len, L, Cnt, Srv, Ss))) >
      {# ( holders <| Srv |<
             establish(subshares(split(Len, L, Cnt, Srv, Ss)))) #}
      ( C <| Srv |<
             contribute(labels(split(Len, L, Cnt, Srv, Ss))))  Conf ]
```

```
if L == label(V) .
```

The syntax `op '{#_#'} : ConfReq -> ConfReq` is used to bracket messages that must be transformed as a unit. A series of replications, simplifications and coalitions transform the one message to `holders` into one message to `holders-of(I)`, for I $= 1, \ldots,$ `Len`, with just the subshares that a server with `I` in its index set should have. `holders` and `holders-of(I)` are of sort `AId`.

When the coordinator has received enough labels, at least one per index, it chooses l (`sharing-length Len` in the specification) of them to specify the subsharings to be used to construct the new sharing, and sends them to all servers. When a server has all the subshares it needs, it constructs its new share set, and sends the coordinator a `computed(L)` message, where `L` is the label with the new version number.

The coordinator must wait for a quorum of $2t+1$ `computed` messages from different servers before concluding that the new sharing has been installed. The replication meta-object for the coordinator collects these messages, and when the quorum is reached, sends down (the message corresponding to)

```
C <| comp-quorum |< computed(L),
```

where `comp-quorum` is of sort `AId`. After updating the current version number and resetting other attributes, the coordinator sends

```
all <| C |< finished(L),
```

which instructs servers to update the current version number and delete old shares and all the subshares.

4.2.2 Replication Layer

As described, some messages have an abstract recipient or sender, representing a set of actual principals. It is the task of the replication meta-object to translate between abstract and concrete messages.

Motivated by this example a somewhat general parameterized replication meta-object can be defined. As configuration requests move up they may be replicated, and as messages come down they may be filtered if recognized to be contributing to a quorum.

```
(oth REPL is including META-STACK .
    sorts ReplicatorMap QuorumMap Filter .
    class Replicator |          replicator-map : ReplicatorMap,
          filter : Filter,      quorum-map : QuorumMap .
    subclass Replicator < MetaObject .
    *** operator declarations
endoth)
```

A replicator map `RM` determines whether the configuration request `CR` should be replicated. Whatever the case, `replications(RM, CR)` reflects the transformation. It may be that some of the messages produced may have the sender as a recipient also. Thus, operations `go-up` and `go-down` select what, if anything, should go in each direction. Some configuration requests moving up may serve as requests that require a quorum of responses. As they go up they modify a filter F, so that when the responses arrive they may be recognized and it can be determined when the quorum is reached.

```
(omod REPLICATION-LAYER[R :: REPL] is
    *** variable declarations
    crl [one-to-many] :
        < RO : Replicator.R |    up : CR CRConf,
            replicator-map : RM, down : Ml,
            out : CRl,           filter : F >
     => < RO : Replicator.R |    up : CRConf,
            out : CRl .*. go-up(replications(RM, CR)),
            down : Ml .*. go-down(replications(RM, CR)),
            filter : construct-filter(F, CR) >
    if not (to-be-u-filtered(F, CR)) .

    crl [many-to-one-from-above] :
        < RO : Replicator.R |    filter : F,
            in : M MConf,        down : Ml,
            quorum-map : QM >
     => < RO : Replicator.R |    filter : d-filter(F, M)
            in : MConf,          down : Ml .*. d-filtered(QM, F, M) >
    if to-be-d-filtered(F, M) .
    *** more rules
endom)
```

As messages move down through the replication layer, the filter F recognizes any message that is a response to be counted towards a quorum. Only when the quorum is reached is a message passed to report the fact.

To define a replication meta-object to be stacked over server and coordinator meta-objects the object module

```
(omod REQUESTS-N-RESPONSES is
    *** module importations
    class Communicator |
        a2c-map : A2C-Map,    responses-quorums : QuorumMap,
        response-recognizer : RespRecognizer .
    subclass Communicator < MetaObject .
    *** operator declarations and definitions
endom)
```

is introduced. An `A2C-Map` is a mapping from `AId` to `SetOfOids`. The messages of the protocol have associated a `MsgType`; a `QuorumMap` maps a `MsgType` to a quorum, a `MachineInt`. A `RespRecognizer` is a set of pairs consisting of an expected response, of sort `Message`, and a `SetOfOids` identifying those that have sent that response. `a2c-map` and `responses-quorums` depend on the parameters n and t of the system; `response-recognizer` is initially empty, and it is built and modified as messages move up and down.

The module `REPLICATION-LAYER[APSS-Messages]`, where `APSS-Messages` is a view from `REPL` to `REQUESTS-N-RESPONSES`, specifies the replication meta-object.

4.2.3 Encryption Layer

The encryption meta-object simply encrypts every outgoing message with the public key of the recipient, and decrypts incoming messages.

4.3 Adversaries and Compromises

This first version of the protocol assumes passive adversaries.

```
class PassiveLinkAdversary | messages : SetOfMsgs .
class PassiveServerAdversary .
subclass PassiveServerAdversary < Server .
```

The link adversary can eavesdrop on links, while the server adversary gets access to local information of a crashed server. Since `PassiveServerAdversary` is not a subclass of `ActiveServer` it may not impersonate its victim. A server compromise is modelled by a rule that replaces the meta-tower representing the server by a `PassiveServerAdversary` object with the state of the server at the time of the crash.

5 APSS with Multiple Coordinators

This second version of the protocol assumes that a coordinator too may crash and be vulnerable to a passive server adversary. A coordinator p and a server p are regarded as processes of the same host; one is compromised if and only if the other is. Under this assumption there must be at least $t + 1$ coordinators, each of which may initiate a run.

The protocol is modified so that one of the competing runs succeeds in installing a sharing. Only the specification of the base layer needs to be modified. Servers act as coordinators, yet both roles are kept as independent as possible. A coordinator in this protocol is of the class `ActiveCoordinatorServer`, a subclass of `ActiveServer` and `ActiveCoordinator`. A sharing label must now identify the coordinator of the run that generated it, and `sharing-label`, that of the current sharing, replaces `current-version-number`. `Server` has an extra attribute, `finished`, to store the latest `finished` message received, which ends a run. When a server receives a message from a coordinator of a competing run, it will forward the stored `finished` message in response. When a coordinator receives a `finished` message it aborts its run, and forwards the message to all servers. One special rule is needed to handle the case when a coordinator has ended or aborted its run and is informing itself, as a server, to end its run.

```
rl [finish-own] :
    [ Conf
      < C : ActiveCoordinatorServer |    run-in-progress : false,
            status : up,                 sharing-label : L,
            shares : Ss,                subshares : SSs,
            runs : (C, L', old-n-new) U Rs,    finished : FM >
        ( C <| C |< finished(L) ) ]
    => [ Conf
        < C : ActiveCoordinatorServer |
            finished : C |< finished(L), runs : emptySetOfRuns,
            shares : Ss >-[ L ],        subshares : SSs >- > ] .
```

6 Defending Against Active-Link Adversaries

The next version of the protocol defends against active link adversaries. These can insert, modify, delete, reorder or replay messages. The most difficult against which to defend is the deletion of messages. This section sketches very briefly elements of the specification of this version.

This version assumes fair links. All messages sent might not be delivered, but if infinitely many are sent, infinitely many are correctly delivered. To approximate reliable links for multicasts [6] introduces the *group-send* primitive. A *group-send*(p, m, ack, d) by p is defined as follows. For any server q, p constructs a message m for q, which it repeatedly sends to q. When q receives the message, it sends back $ack(m)$ to p. This *group-send* terminates when p receives d acknowledgments, or when it receives a `finished` message from another run. For example, the `init` message sent by the coordinator to all servers (see page 6) is acknowledged by `contribute` messages. Here identical messages are sent to all servers. This primitive is also used when messages of the same type are tailored for each recipient, as the `establish` messages sent to all servers are (see page 6), which require $t + 1$ acknowledgments.

The specification of this protocol requires modifications to the base and replication layers of the last version. New protocol messages are introduced to serve as acknowledments, and the replication layer must allow for the resending of messages. The filter in the replication layer, must not only recognize acknowledgments, but also have a timer per message group, and store the messages, so they can be resent if necessary. Ideas for the treatment of time for object-oriented systems in [5] and the use of timers in [3] can be used.

7 Correctness Criteria

The asynchronous proactive secret-sharing protocol has three correctness requirements: *secrecy*, the secret remains unknown to adversaries; *availability*, correct servers collectively have enough shares to reconstruct the secret; and *progress*, every run eventually terminates, meaning that the correct servers eventually delete the shares of the previous run.

Predicates for availability and secrecy are defined as follows:

```
op  the-secret            : -> Secret .
ops is-available is-secret : Configuration -> Bool .
```

```
op  all-secrets            : SetOfShares -> SetOfSecrets .
eq is-available(Conf)
     = the-secret in all-secrets(servers-shares(Conf)) .
eq is-secret(Conf)
     = not(the-secret in all-secrets(adversaries-shares(Conf))) .
```

`servers-shares(Conf)` is the set of all shares and subshares held by active servers; `adversaries-shares` is the analogous set for adversaries. There are three representations for sharings, depending on whether a sharing was obtained from a split; from the refreshment process; or was installed at the end of a run. `all-secrets` transforms a set of secrets by recursively recovering secrets from sharings.

8 Search Strategy for a Layered Specification

The availability and secrecy correctness criteria of the APSS protocol are violated if there exists some run of the protocol in which at some point the servers cannot recover the secret or the adversaries can. Maude is a reflective language that supports an arbitrary number of metalevels. Any specification can be meta-represented at the next metalevel, and thus manipulated from it. In particular, Maude can be used to specify strategies, specifications to control the rewriting process. This section considers some of the properties of the specification of the APSS protocol. It motivates and describes a strategy that exhaustively generates all computations of a specification with such properties, which can be used to formally analyze the protocol by searching for runs that violate the availability or secrecy property.

8.1 Exhaustive Search

The complete specification of a protocol consists of the specification of the protocol itself, the actions and messages of the intended participants, augmented by the specification of the environment in which it must operate, and of the properties it guarantees. All three components can be specified by rewrite theories, with a property being specified negatively by rules that detect and in some way capture a state that violates it. A computation is a sequence of rewrite rule applications. Then, a protocol can be formally analyzed by generating all computations from a given initial state, and searching for those that end with application of some rule that specifies the violation of a guaranteed property.

8.2 Simplification for a Benign Environment

APSS is parameterized by n, the number of servers, and t, the threshold of compromises. A particular (n, t) instantiation of APSS is formally analyzed by an exhaustive search of the computation tree rooted in the initial state for the instantiation. In this state each server holds a standard share s_i, of some initial sharing of the maintained secret, for each i in its index set. For the first version of the protocol there are n servers and one coordinator. For the second, there are n principals, of which at least $t+1$ are servers that act as coordinators. The initial state has neither adversaries, nor messages. Thus, for both versions the initial state is a configuration of three-meta-object towers.

Both protocols assume a benign environment in which adversaries may only access local information from crashed servers. Under this assumption encryption is unnecessary. Thus, the search space may be reduced somewhat by having initial states with two-meta-object towers, with no encryption meta-objects.

8.3 Canonical Form of Computations

A computation is a sequence of transitions, which for a system specified in Maude are rewrite rule applications. In concurrent systems some transitions are incomparable, resulting in some spuriously distinguished computations. The problem with an exhaustive search is that of combinatorial explosion. Therefore, it is important to minimize these artificial distinctions.

Definition 1 *Transitions τ and τ' are incomparable if whenever $s \xrightarrow{\tau} t$ and $s \xrightarrow{\tau'} t'$ there exists a state s' such that $t \xrightarrow{\tau'} s'$ and $t' \xrightarrow{\tau} s'$.*

$$\begin{tikzcd}
 & s \arrow[dl, "\tau"'] \arrow[dr, "\tau'"] & \\
t \arrow[dr, "\tau'"'] & & t' \arrow[dl, "\tau"] \\
 & s' &
\end{tikzcd}$$

Thus, transitions that change disjoint substates are incomparable.

A meta-object has the four communication attributes `in`, `out`, `up` and `down`, and possibly other attributes. An internal transition of a meta-object changes only the state of that meta-object. For secret-sharing and replication meta-objects, the internal transitions either consume a single element of one of the importing attributes (`in`, `up`), or produce new elements of its exporting attributes (`down`, `out`). Communication transitions, which are applications of rules `in`, `out`, `up` and `down` (see [2]), are complementary to internal meta-object transitions with respect to the communication attributes: the ones produce where the others consume, and *vice versa*. Applications of rules `up` and `down` transfer data between adjacent meta-objects in a meta-stack; `up`, from attribute `out` to attribute `up`; `down`, from attribute `down` to attribute `in`. Applications of rules `in` and `out`, in turn, transfer data between the top meta-object of a meta-object tower and the surrounding configuration; `in`, from the configuration into the `in` attribute; `out`, from the `out` attribute to the configuration.

Proposition 1 *Let i, i_o, $i_{o'}$ be internal transitions of secret-sharing or replication meta-objects, where the subscripts identify a meta-object and $o' \neq o$.*

Let c, c' be communication transitions. Then

$$\begin{tikzcd} & {} \arrow[dl, "i"'] \arrow[dr, "c"] & \\ {} \arrow[dr, "c"'] & & {} \arrow[dl, "i"] \\ & {} & \end{tikzcd} \qquad \begin{tikzcd} & {} \arrow[dl, "c"'] \arrow[dr, "c'"] & \\ {} \arrow[dr, "c'"'] & & {} \arrow[dl, "c"] \\ & {} & \end{tikzcd} \qquad \begin{tikzcd} & {} \arrow[dl, "i_o"'] \arrow[dr, "i_{o'}"] & \\ {} \arrow[dr, "i_{o'}"'] & & {} \arrow[dl, "i_o"] \\ & {} & \end{tikzcd}$$

To reduce the search space a canonical order is imposed on some of the incomparable transitions. Let *ss* and *r* denote internal secret-sharing and replication transitions, and *in*, *out*, *up* and *down* denote applications of rules `in`, `out`, `up` and `down`. Then

$$in\ out\ r \prec up \prec down \prec ss$$

The initial state is assumed to have all communication attributes of all meta-objects empty. Therefore, this order amounts to a policy of immediate delivery.

8.4 APSS Replication Layer is Church-Rosser

The behavior of the replication layer is described by four conditional rules. Two consume upward-moving messages; two, downward-moving messages; and in each pair the condition of one rule is the negation of the condition of the other. Thus, any message consumed by a replication meta-object determines a unique transition. The application of a replication rule may produce zero or more messages to be exported up, down or in both directions, enabling `out` and `down` transitions, with the `out` transitions enabling `in` transitions by other replication meta-objects. In an APSS instantiation of the parameterized replication layer only the `Filter` attribute may have its value changed. It is implemented as a set of tuples. An upward-moving message may add a new tuple to the filter. A downward-moving message may replace a tuple, just to change a component that is a count of downward-moving messages. Since importing attributes (`in` and `up`) are multisets, and downward-moving messages may only change the value of a count, internal transitions of a meta-object are incomparable.

Proposition 2 *Let r_o and r'_o be internal transitions of an APSS replication meta-object o. Then*

$$\begin{tikzcd} & {} \arrow[dl, "r_o"'] \arrow[dr, "r'_o"] & \\ {} \arrow[dr, "r'_o"'] & & {} \arrow[dl, "r_o"] \\ & {} & \end{tikzcd}$$

From Propositions 1 and 2 it follows by induction that an APSS replication layer is Church-Rosser.

Proposition 3 *Let $s \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_k} t$ and $s \xrightarrow{\tau'_1} \cdots \xrightarrow{\tau'_l} t'$ be sequences of transitions that change the state of APSS replication meta-objects. Then there exist state s' and sequences $t \xrightarrow{\tau_{k+1}} \cdots \xrightarrow{\tau_n} s'$ and $t' \xrightarrow{\tau'_{l+1}} \cdots \xrightarrow{\tau'_m} s'$.*

8.5 Hiding a Layer

The availability and secrecy properties are predicates on the state of the secret-sharing layer. Therefore, the computation tree that needs to be explored is the one whose nodes are states of the secret-sharing layer. The transitions of the replication layer should be hidden.

Given a state of the secret-sharing layer, a child in the computation tree is obtained by the application of an internal secret-sharing rule, or the immediate delivery of a message sent, which hides the replication transitions. Since the replication layer is Church-Rosser, it is convenient to use the function `meta-rewrite` of the predefined module META-LEVEL. This function takes a `Module` M, a `Term` T and an integer n, and uses a default, fair strategy to obtain a `Term` T'. For $n = 0$, the resulting T' cannot be further rewritten. In order to provide `meta-rewrite` with an appropriate module for this purpose the specification is modified as described below.

8.6 Configuration Layers

As specified, the state of an APSS instantiation is a configuration that has two-meta-object towers. Since the transitions that move messages up and down the towers, and in and out of the configuration should be hidden during the exhaustive search, the meta-object towers are disassembled. For the search the state is represented by a replication configuration and a secret-sharing configuration. The replication configuration has the replication meta-objects. The secret-sharing configuration has the adversary objects and the secret-sharing meta-objects for servers and coordinators. A node in the computation tree is a term of the form $T_{ss} \sharp T_r$, where T_{ss} and T_r are meta-representations of the secret-sharing and replication configurations.

T_r remains unchanged by internal secret-sharing transitions. The hidden transitions are triggered when a message is sent. The rule `up` is replaced by a pair of rules: `up>`, which initiates the transfer of a configuration request from the secret-sharing configuration by exporting it from a meta-object into the configuration. A counterpart rule `<up` for the replication configuration imports it from the configuration into a meta-object. New syntax is introduced for the inter-configuration transfer, which pairs the configuration request with an object id to identify matching meta-objects in both configurations. The application of these matching rules is mediated by a metalevel operation that extracts this message from the secret-sharing configuration and inserts it into the replication configuration. Similarly, rule `down` is replaced by the replication layer rule `<down` and the secret-sharing layer `down>`, which are mediated by a metalevel operation to transfer any messages being sent from the replication configuration to the secret-sharing configuration. Matching applications of `out` and `in` are replaced by applications of a new rule `<out<in`.

Given modules M_{ss} and M_r that specify the secret-sharing and replication layers of the original specification, modules for the analysis can be obtained from them by including the new communication rules: $M^*_{ss} = M_{ss} +$ (`up>`, `down>`), $M^*_r = M_r +$ (`<up`, `<down`, `<out<in`) $-$ (`<up`, `<down`, `<in`, `<out`). Some details are given in the Appendix.

9 Conclusion

Some versions of the asynchronous proactive secret-sharing protocol have been described. The specification has been factored into three layers: secret-sharing, replication and encryption. A general parameterized layer for the replication of messages and collection of quorums of responses was instantiated for an (n, t) instantiation of the APSS protocol. A strategy that exploits the structure of the specification to reduce the search space was developed.

Thousands of complete runs of length 40 are found, none shorter. Since the period of vulnerability is the duration of two consecutive runs, a search tree should be at least of depth 80. Thus, further ways of reducing the space search will be sought. This work was done using Maude 1.0.5. Continuing work will exploit the more powerful features of newer versions of Maude in better analysis techniques and a more complete specification of the protocol.

A Appendix

Syntax for inter-configuration messages

```
op up`(_`)>_ : ConfReq Oid -> ConfReq .
op down`(_`)>_ : Msg Oid -> Msg .
```

Communication rules for the secret-sharing configuration:

```
rl [up>] :
    [ Conf  < O : MetaObject | out : CR .*. CRl, base : B  > ]
 => [ Conf  < O : MetaObject | out : CRl >    (up( CR ) > B) ] .

rl [down>] :
    [ Conf  < O : MetaObject | in : MConf, base : B >
      (down( M ) > B) ]
 => [ Conf  < O : MetaObject | in : MConf M > ] .
```

Communication rules for the replication configuration:

```
rl [<down] :
    [ Conf  < O : MetaObject | down : M .*. Ml, base : B > ]
 => [ Conf  < O : MetaObject | down : Ml >   (down(M) > B) ] .
rl [<up] :
    [ Conf  < O : MetaObject | up : CRConf, base : B >
      (up( CR ) > B) ]
 => [ Conf  < O : MetaObject | up : CRConf CR > ] .

rl [<out<in] :
    [ Conf  < O  : MetaObject | base : B, in : MConf   >
```

```
           < O' : MetaObject | out : (B <| M) .*. CRl > ]
 => [ Conf  < O  : MetaObject | base : B, in : MConf M >
           < O' : MetaObject | out : CRl > ] .
```

Syntax for the meta-representation of the state as a pair of configurations.

```
protecting TUPLE(2)[Term, Term]
    * (sort Tuple[Term, Term] to Layered-Term,
       op `(_`,_`) to _#_,
       op p1_ to ss._ ,
       op p2_ to r._ ) .
subsort Layered-Term < Term .
```

Given a state T, the state resulting from the application of an internal secret-sharing transition specified by a rule labelled by L:

```
        extTerm(meta-apply(Mss, ss. T, L, none, 0))  #  r. T.
```

Operations for the extraction of objects, up and down messages from configurations:

```
ops o^_ u^_ d^_ : Term -> Term .
```

Operation for combining configurations:

```
op  ^ : -> Term .
op  _&_ : Term Term -> Term [assoc comm id: ^] .
```

Implementation of immediate delivery:

```
eq deliver(Mss, Mr, T) = r>ss(Mss, r(Mr, ss>r(Mss, Mr, T))) .
eq ss>r(Mss, Mr, T)
   = o^(extTerm(meta-apply(Mss, ss. T, 'up>, none, 0)))
     # ( r. T
         & u^(extTerm(meta-apply(Mss, ss. T, 'up>, none, 0)))) .
eq r(Mr, T) = ss. T  #  meta-rewrite(Mr, r. T, 0) .
eq r>ss(Mss, T)  =  d>>(Mss, (ss. T) & (d^ r. T))  #  o^ r. T .
eq d>>(Mss, T)
   = if extTerm(meta-apply(Mss, T, 'down>, none, 0)) == error*
          then T
          else d>>(Mss, extTerm(meta-apply(Mss, T, 'down>, none, 0)))
      fi .
```

References

[1] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and programming in rewriting logic.*, SRI International, 1999. Available at `http://maude.csl.sri.com/`.

[2] G. Denker, J. Meseguer, and C. Talcott. *Rewriting semantics of meta-objects and composable distributed services*, Electronic Notes in Theoretical Computer Science, **36** (2000). Available at `http://www.elsevier.nl/locate/entcs/volume36.html`.

[3] I. Mason and C. Talcott. *Simple network protocol simulation within Maude*, Electronic Notes in Theoretical Computer Science, **36** (2000). Available at `http://www.elsevier.nl/locate/entcs/volume36.html`.

[4] J. Meseguer. "A logical theory of concurrent objects and its realization in Maude." In G. Agha, P. Wegner and A. Yonezawa, editors, "Research Directions in Concurrent Object-Oriented Programming," pages 314–390. The MIT Press, 1993.

[5] P. Ölveczky and J. Meseguer. *Real-Time Maude: A tool for simulating and analyzing real-time and hybrid systems*, Electronic Notes in Theoretical Computer Science, **36** (2000). Available at `http://www.elsevier.nl/locate/entcs/volume36.html`.

[6] L. Zhou. "Towards fault-tolerant and secure on-line services," Ph.D. thesis, Cornell University, 2001. Available at `http://www.cs.cornell.edu/Info/People/ldzhou/coca.htm`.

[7] L. Zhou, F. B. Schneider, and R. van Renesse. *COCA: A secure distributed on-line certification authority*, ACM Transactions on Computer Systems. To appear. Available at `http://www.cs.cornell.edu/Info/People/ldzhou/coca.htm`.