Rewriting Semantics of Meta-Objects and Composable Distributed Services*

G. Denker and J. Meseguer
Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA
{denker,meseguer}@csl.sri.com

C. Talcott
Computer Science Department, Stanford University, Stanford, CA 94305, USA
clt@cs.stanford.edu

March 1, 1999

Contents

*Supported by DARPA through Rome Laboratories Contract F30602-97-C-0312, by DARPA and NASA through Contract NAS2-98073, by Office of Naval Research Contract N00014-96-C-0114, and by National Science Foundation Grants CCR-9505960 and CCR-9633363.

1 Introduction

In distributed computing and in communications software there is a great interest in modular and composable approaches. In particular, services such as security or fault-tolerance and services intended for boosting performance that may in practice be "hard-wired" across different parts of the code of a distributed system should be treated in a much more modular way, so that they can be dynamically added to a system, and so that several such services can be composed to obtain their combined benefits. Besides the good software engineering reasons of thus making the systems much more easily reusable and adaptable, modularity and composability are also crucial at runtime, in the sense that some of these services, that may have a cost in performance, should not be used always and everywhere. Instead, they should be installed dynamically and selectively at runtime, in those areas or domains of the distributed system where they are needed in response to some security threat, failure, need for boosting performance, and so on.

A number of authors, in the areas of distributed computing and networks (see for example papers by Agha (cited below) and [17, 11, 19]), and also researchers in formal methods (c.f. [29, 31, 27]), have proposed some concepts and solutions to deal with the problems posed by composition of distributed services. The notion of "weaving" [15] is also a mechanism that could be used to deal with some of these problems. There has also been some work on modular specification and reasoning (see below). Several of these approaches use reflection to achieve modularity and adaptability. But, all of these approaches recognize that the goal of achieving truly modular and composable distributed services, and ensuring good properties in compositions of such services is quite subtle.

To begin with, a satisfactory formal semantics of what is meant by composable distributed services has not yet been given, nor have the reflective aspects of such compositions been adequately formalized. Chandy and Misra [6] propose a " rely-guarantee" discipline for treating global and local properties of a system separately, and Abadi and Lamport [1] propose a method for describing open components of concurrent systems using assumption/guarantee assertions [14]. However, the above approaches address what might be called horizontal composition of systems. We are not aware of any systematic approach to the study of properties of services composed in a reflective and vertical way and based on the semantics of such compositions. This paper proposes a semantic approach to make precise the reflective concept of composable service in a distributed system, and to reason about the properties of service compositions. Our approach is based on the executable formal semantics for distributed object-oriented systems provided by rewriting logic [25] and explicitly addresses the reflective properties that are essential for having a truly modular notion of service. The formal model that we present seems promising not only for formal analysis and symbolic simulation, but also for the application of theorem proving methods to formally verify important properties of services and their compositions. We have been strongly influenced by the onion skin model of actor reflection developed by Agha and his collaborators [3, 13, 34, 12, 2, 33, 4] which we generalize and formalize in this paper within the framework of rewriting logic.

Plan

The remainder of this paper is organized as follows. In the rest of this section we informally discuss the onion skin model (§1.1) and illustrate it with examples (§1.2) to motivate the generalization and formalization of these ideas in the main body of the paper. In §1.3 we address in more detail the reasons why semantics and formal methods are important to clarify the concept of composable distributed service and to reason about the composition of such services, and we sketch the main advantages of our approach. In §2 we present a rewriting semantics of distributed meta-objects. We begin with a review of the rewriting logic axiomatization of concurrent objects and specialize this to distributed asynchronous objects in the spirit of the actor model (§2.1). In §2.2 we then describe the representation of meta objects and encapsulation of layers of meta-objects in towers. Finally, we show how distributed objects can be transformed to equivalent single object towers that are then suitable for installation of additional meta-layers (§2.3). In §3 we discuss how to model composable communication services as distributed meta-objects and illustrate the basic ideas with a case study treating fault-tolerance (§3.1), encryption and authentication services (§3.2) and their composition (§3.3). Formal specification of service guarantees and combination of guarantees for composed services are also discussed. Conclusions and future work are discussed in §4.

1.1 The Onion Skin Model of Actor Reflection

The objective of the onion skin model is to reflect on the basic actor primitives of communication (sending and receiving messages) and actor creation while maintaining a balance between the power of the reflective capabilities and the encapsulation properties of the actor model.

In this model each actor has a meta-actor that defines the semantics of its primitive actions. For example, a message send by the base actor becomes a request to its meta-actor to transmit the message. Dually, messages delivered to the actor are received by the meta-actor giving it the ability to control the receive semantics. If no explicit meta-actor behavior is defined, the underlying system semantics provides a default meta-actor behavior. An actor together with its meta-actor appears to the outside like a normal actor, and thus can be controlled by a further meta-level actor. This gives rise to layers of meta-levels and hence the "onion skin" analogy. Distributed services for a group of actors can be expressed in terms of meta-actor layers that coordinate to achieve some overall property. This allows different protocols for security, reliability, quality of service and other services to be composed and modified dynamically as the system runs and new needs arise by adding new meta-levels.

As proposed by Agha et al., the onion skin model provides a conceptual foundation for separation of concerns in developing and (dynamically) adapting open distributed systems to meet changing environment conditions and requirements. For example, application functionality, communication protocols and security requirements, and failure / dependability semantics, can be treated separately and modularly as independent layers providing services that may be composed in various ways to achieve a desired overall behavior. This model has been used to support a number of high-level declarative programming abstractions such as synchronizers, activators, real-time synchronizers, actor spaces, and protocols that abstract over interaction patterns. Several prototype languages have been developed based on these ideas [12, 33, 30].

1.2 Example of Composing Services

In the onion skin architecture the implementation of a service is described by an abstract protocol that specifies a set of roles to be played by participants, the requirements on role players, and installation information. When a protocol is installed, each component is encapsulated in a layer that implements the role it is to play. The added role behavior only affects communication between participants of the protocol instance. No knowledge of the component or modification of the component (actor) itself is needed. Of course, for the protocol to implement the service correctly it may be necessary that the components satisfy some specific interface or behavioral requirements.) To illustrate how these mechanisms might be used we sketch an example scenario.

Consider two components A and B (Figure 1) where B provides some service used by A (perhaps a data repository).

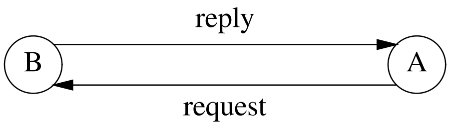

Figure 1: A sends requests to B, and B replies and perhaps changes internal state.

Now, some agent (possibly A, possibly not) decides that B needs a backup B' and installs a backup protocol. To be concrete, suppose the protocol has three roles: client, server, and backup. Playing the server role requires: (1) sending copies of all requests from A to the backup, as well as to the server; (2) periodically taking a snapshot of the state of B and sending this to the backup as an update; and (3) adding a tag to replies to the client, for use in failure recovery. Playing the client role requires removing the added tags before passing replies on to A (and keeping some record). Playing the backup role requires collecting the messages between updates, installing updates, and sending messages accumulated between updates to B' in the case that B' must take over. The result of installing the protocol is shown in Figure 2. A clone, B', of B has been created on some independent processor. Roles have been installed on the participants: A as client; B as

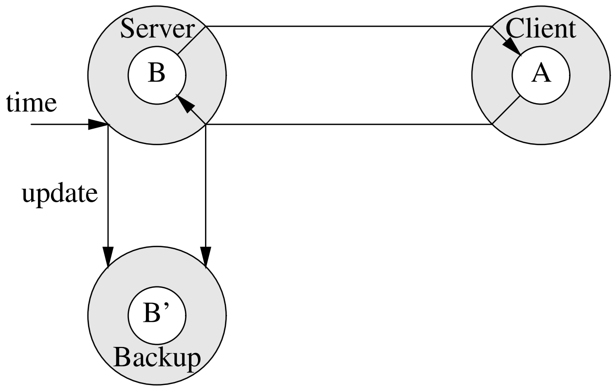

Figure 2: Backup service governing interactions of A and B, with A playing the client role, B playing the server role, and a new agent B’ playing the backup role.

server; B’ as backup. A timer has also been created to notify the server when an update is needed. Note that, since roles only affect communication between participants, messages between A and components other than B are passed through without any effect.

Now, A starts communicating sensitive information to B. To protect the information, an encryption service is put in place by installing encryptor roles on A and B. These roles take care of encrypting outgoing messages and decrypting incoming messages (Figure 3). Notice that enforcement of the encryption service is independent of the backup activity: players of the backup roles need not have any knowledge of the encryption going on.

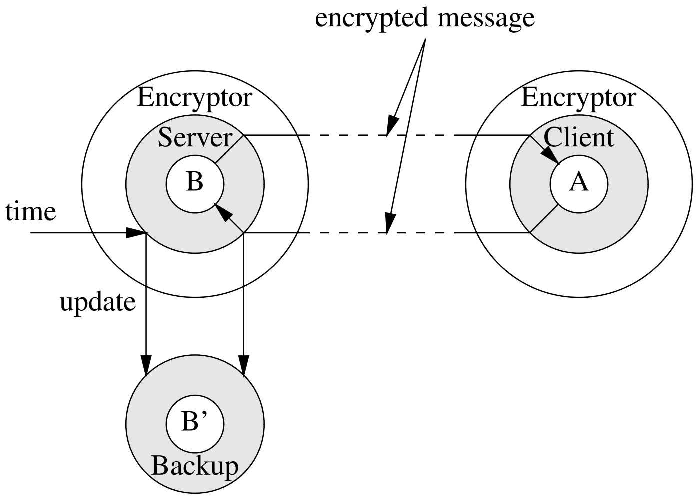

Figure 3: Messages between A and B are encrypted and decrypted by the encryption layers on each.

Finally, the communication medium between B and B’ becomes flaky and a checksum protocol is installed between the server and the backup. This is easily done, and again this is independent of the encryption of communication between B and A, even though B is now playing several roles.

The onion skin model focuses on what we might call vertical composition of services. Although it is easy to install new protocol layers, there is still a need to understand the possible interactions of the layers. For example, if an encryption layer were installed upon the checksum layer this would probably render the checksum useless, while installing the layers in the other order would work as expected. In either case the underlying application remains blissfully ignorant of any service layers and continues to function as before.

1.3 Why Do We Need Semantics And Formal Methods?

Although a number of the languages based on actor reflection have been given a rigorous mathematical semantics [12, 33, 30], it seems fair to say that up to now the layered meta-actor model has not been given a satisfactory language-independent formal semantics. Furthermore, as we explain later in the paper, we need to extend the semantic treatment from actors to more general distributed objects to formalize both the reflective semantics of meta-objects and the behavior of the hostile environments that many of the services are designed to protect a distributed system against. Moreover, the reflective notion of meta-object *composition* is indeed the key notion that needs to be given a precise formal semantics.

Our rewriting logic formalization of meta-objects gives a precise account of the semantics of meta-object composition. We show how a system of concurrent objects communicating through asynchronous message passing can be naturally transformed into a behaviorally equivalent system of *meta-object towers of height one*. Composition of meta-objects is very simply expressed by stacking meta-objects on top of each other, with the application meta-object at the bottom of the stack. The behavior of meta-object compositions is then formalized by general rewrite rules governing the communication between the different levels of a stack of meta-objects, as well as by the specific additional rules that govern the behavior of meta-objects providing a particular service.

Another important topic in need of a systematic treatment is the issue of the formal properties or *guarantees* offered by the use of a given distributed service. Furthermore, there is a very strong interest in having methods to reason formally about how and when guarantees offered by different services can be *preserved* when the services in question are combined.

For services protecting against attacks or faults that may be present in the distributed system's communication medium, making formally precise certain guarantees requires first of all making precise the types of *hostile environments* against which the given service guarantees certain desired properties. In our rewriting logic semantics this is achieved by giving a formal specification of such an hostile environment by means of a rewrite theory specifying the hostile behavior. The guarantees in question are then formal properties satisfied by the overall system axiomatized by a union of rewrite theories, namely, by the theory of the application meta-objects on top of which the service has been composed, union with the theory of the hostile environment. In fact, such guarantees are typically *inductive properties* satisfied by the *initial model* of the combined theory; this is similar to the approach by Paulson [28] and Dutertre and Schneider [10], who use inductive methods to prove properties of security protocols. Therefore, the initial model semantics of rewriting logic [23] provides the mathematical basis for the formal verification of such guarantees or properties, and can serve as a basis for justifying the correctness of specific theorem proving techniques developed for such a purpose.

Our modular approach to service composition yields also a parametric and compositional method for combining the properties or guarantees of each service in a given composition. As already pointed out, such service guarantees are not always preserved in a composition: preservation may crucially depend on the composition order. But if they *are* preserved, our semantics makes precise the form in which the properties are instantiated and combined for each composition.

2 Rewriting Semantics of Distributed Meta-Objects

We model a distributed system configuration as a soup (multi-set) of concurrent objects and messages that behave according to a set of rewrite rules describing the behavior of individual objects. We specialize the rewriting logic object model [18, 22, 26] to the case of asynchronous message passing. We call these *distributed objects*. We then introduce meta-objects, which can be layered to form towers that appear to their environment just like distributed objects. The communication between a distributed object and its meta-object is synchronous. This is achieved by giving each meta-object four message queues: `out` and `in` queues for sending messages to and receiving messages from its upper level (another meta-object or the environment); and `up` and `down` queues for interaction with the lower level.

We first explain how concurrent object systems are axiomatized in rewriting logic. We identify the subclass of object rewrite theories axiomatizing asynchronous actor-like behavior, called *distributed objects*.

Since we are interested in modelling protocols and services for achieving various security and fault tolerance properties, we relax the usual actor semantics requirements for reliable point-to-point message delivery and unguessable actor names (see [5, 35, 36]). This allows us to model both the bad and the good guys in the environment (§2.1). We define a subclass of concurrent objects called *meta-objects* and show how meta-objects can be composed to form a layered concurrent object, called a *meta-object tower* (§2.2). We define a mapping of ordinary distributed objects to towers of height one that are behaviorally equivalent to their corresponding objects (§2.3).

2.1 Distributed Concurrent Objects in Rewriting Logic

In general, a rewrite theory is a pair $\mathcal{R} = ((\Omega, \Gamma), R)$, with (Ω, Γ) an equational specification with signature of operations Ω and set of equational axioms Γ; and with R a collection of labelled rewrite rules. The equational specification describes the *static* structure of the system's state space as an algebraic data type. The *dynamics* of the system are described by the rules in R, that specify local concurrent *transitions* that can occur in the system axiomatized by $\mathcal{R}$, and that can be applied *modulo* the equations Γ.

Let us then begin by explaining how the state space of a concurrent object system can be axiomatized as the initial algebra of an equational theory (Ω, Γ). That is, we explain the key state-building operations in Ω and the equations Γ that they satisfy. The concurrent state of an object-oriented system, often called a *configuration*, has typically the structure of a *multiset* made up of objects and messages. Therefore, we can view configurations as built up by a binary multiset union operator, which we can represent with empty syntax (i.e., juxtaposition) as $\_\,\_ : \mathbf{Conf} \times \mathbf{Conf} \longrightarrow \mathbf{Conf}$. (Following the conventions of mix-fix notation, we use underbars "_" to indicate argument positions.) The multiset union operator $\_\,\_$ is declared to satisfy the structural laws of associativity and commutativity and to have identity $\emptyset$. Objects and messages are singleton multiset configurations, and belong to subsorts $\mathbf{Object}, \mathbf{Msg} < \mathbf{Conf}$, so that more complex configurations are generated out of them by multiset union.

An *object* in a given state is represented as a term $\langle O : C \mid a_1 : v_1, \ldots , a_n : v_n \rangle$, where O is the object's name or identifier, C is its class, the a_i's are the object's *attribute identifiers*, and the v_i's are the corresponding *values*. The set of all the attribute-value pairs of an object's state is formed by repeated application of the binary union operator $\_\,,\_$ which also obeys structural laws of associativity, commutativity, and identity; i.e., the order of the attribute-value pairs of an object is immaterial. Particular systems are axiomatized by providing additional operations and equations, specifying, for example, the data operations on attribute values and the structure of messages.

The associativity and commutativity of a configuration's multiset structure make it very fluid. We can think of it as "soup" in which objects and messages float, so that any objects and messages can at any time come together and participate in a concurrent transition corresponding to a communication event of some kind. In general, the rewrite rules in R describing the dynamics of an object-oriented system can have the form

$$
\begin{array}{rl}
r(\bar{x}) : & M_1 \ldots M_n \ \langle O_1 : F_1 \mid \mathit{atts}_1 \rangle \ldots \langle O_m : F_m \mid \mathit{atts}_m \rangle \\
\longrightarrow & \langle O_{i_1} : F'_{i_1} \mid \mathit{atts}'_{i_1} \rangle \ldots \langle O_{i_k} : F'_{i_k} \mid \mathit{atts}'_{i_k} \rangle \\
& \langle Q_1 : D_1 \mid \mathit{atts}''_1 \rangle \ldots \langle Q_p : D_p \mid \mathit{atts}''_p \rangle \\
& M'_1 \ldots M'_q \\
& \mathit{if}\ C
\end{array}
$$

where r is the rule's label, $\bar{x}$ is a list of the variables occurring in the rule, the Ms are message expressions, $i_1, \ldots , i_k$ are different numbers among the original $1, \ldots , m$, and C is the rule's condition. That is, a number of objects and messages can come together and participate in a transition in which some new objects may be created, others may be destroyed, and others can change their state, and where some new messages may be created. If two or more objects appear in the lefthand side, we call the rule *synchronous*, because it forces those objects to jointly participate in the transition. If there is only one object in the lefthand side, we call the rule *asynchronous*. A concurrent object rewrite theory is called a *distributed object theory* if the rules are asynchronous, with exactly one object and at most one message on the lefthand side. In addition, message expressions must have the form $O \triangleleft M$, where O is an object identifier and M is a term representing the information to be communicated. We call messages of this form *message packets*. A distributed object configuration is then a multi-set of objects and message packets in the context of a distributed object rewrite theory.

Rewriting logic then gives an inference system [23] to deduce, for a system axiomatized by a rewrite theory $\mathcal{R}$, which are all the finitary concurrent computations possible in such a system. Such computations are identified with *proofs* of the general form $\alpha : C \longrightarrow C'$ in the logic, where C and C' are terms representing system configurations.

2.2 Meta-Objects and Towers

Meta-objects are special concurrent objects that implement individual layers of layered objects, called *meta-object towers*. Each layer serves as the meta-object for the layered object below, interpreting requests from the layer below to send messages and create new objects, and requests from the layer above or the environment for delivering messages.

All meta-objects belong to a subclass of the class `MetaObject`, an object class with five distinguished attributes: `in`, `dwn`, `out`, `up`, `base`. The attribute `in` (respectively, `dwn`) is a queue of incoming messages representing messages to be delivered from the layer above or from the outside (respectively, to the layer below). The attribute `out` (respectively, `up`) is a queue of message configurations—that is, multi-sets of message packets—consisting of outgoing (respectively, upgoing) requests for message transmittal and object creation. The attribute `base` is an object identifier corresponding to the identity of the object at the bottom of the stack. We further require that meta-object rules do not explicitly consume or create configuration elements. They may only remove items from the `in` and `up` queues and place items in the `out` and `dwn` queues and modify other attributes. To interact with other meta-objects or with the environment a meta-object must be part of a meta-object tower. A meta-object tower is a structure of the form

$$\{ \langle\, O_k : MC_k \mid atts_k, \mathtt{base} : O_0 \,\rangle \circ \ldots \circ \langle\, O_0 : MC_0 \mid atts_0, \mathtt{base} : O_0 \,\rangle \}$$

built by means of an associative stack concatenation operation `_○_` with identity `nil`, and where the rightmost object (with identifier O_0) is the base layer of the stack, and the leftmost object (with identifier O_k) is the top layer of the stack. Each class MC_i is a subclass of `MetaObject`, and the value of `base` in each element is O_0. The following rewrite rules explain how a layer of a meta-object tower communicates with the layers above and below it, and, in the case of the top layer, how the meta-object tower communicates with the environment.

$$\begin{array}{ll} \text{in:} & O_0 \triangleleft M \;\{ \langle\, O_k : MC_k \mid \mathtt{base} : O_0, \mathtt{in} : Q \,\rangle \circ MOS \,\} \\ & \quad \longrightarrow \{ \langle\, O_k : MC_k \mid \mathtt{base} : O_0, \mathtt{in} : Q \cdot M \,\rangle \circ MOS \,\} \end{array}$$

$$\begin{array}{ll} \text{out:} & \{ \langle\, O_k : MC_k \mid \mathtt{out} : mCf \cdot cQ \,\rangle \circ MOS \,\} \\ & \quad \longrightarrow \{ \langle\, O_k : MC_k \mid \mathtt{out} : cQ \,\rangle \circ MOS \,\} \;\; CreateConf(mCf) \end{array}$$

$$\begin{array}{ll} \text{up:} & \langle\, O' : MC' \mid \mathtt{up} : cQ' \,\rangle \circ \langle\, O : MC \mid \mathtt{out} : mCf \cdot cQ \,\rangle \\ & \quad \longrightarrow \langle\, O' : MC' \mid \mathtt{up} : cQ' \cdot mCf \,\rangle \circ \langle\, O : MC \mid \mathtt{out} : cQ \,\rangle \end{array}$$

$$\begin{array}{ll} \text{down:} & \langle\, O' : MC' \mid \mathtt{dwn} : M \cdot Q' \,\rangle \circ \langle\, O : MC \mid \mathtt{in} : Q \,\rangle \\ & \quad \longrightarrow \langle\, O' : MC' \mid \mathtt{dwn} : Q' \,\rangle \circ \langle\, O : MC \mid \mathtt{in} : Q \cdot M \,\rangle \end{array}$$

where MOS, MOS' are possibly empty lists of meta-objects, Q, Q' are queues of messages cQ, cQ' are queues of message configurations, M is a message, mCf is a message configuration containing requests for creation of a meta-object configuration, and in the 'out' rule $CreateConf(mCf)$ is the meta-object configuration whose creation is requested by mCf. Figure 4 shows a meta-object tower with base object O, message packet $0 \triangleleft M$ coming in from the surrounding environment, configuration $CreateConf(mCf)$ being created, message M' ready to move from the 'down' queue of meta-object MO to the 'in' queue below, and creation request mCf' ready to move from the 'out' queue of MO to the 'up' queue above (possibly to the environment).

Note that by treating a meta-object stack with base object identifier O as an object with identifier O, meta-object rewrite theories can be considered as distributed object rewrite theories, and object and meta-object towers can co-exist and interact, without needing to know what flavor of distributed objects are present in the system.

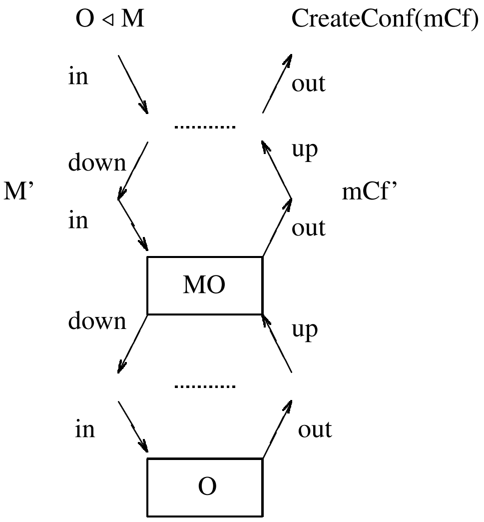

Figure 4: A meta-object tower

2.3 Transforming Distributed Objects into Meta-Objects

We define a mapping from distributed object rewrite theories to meta-object rewrite theories, and show that this mapping preserves behavior. To simplify the presentation we make use of the simplifying assumption that a distributed object is always enabled to receive any message under consideration. This is not a real restriction (see for example [20, 21]).

We assume an object theory in which the attribute identifiers of object classes are distinct from the distinguished meta-object attribute identifiers. To each distributed object class OC we associate a meta-object class $o2m(OC)$. $o2m(OC)$ is a subclass of `MetaObject` and has as additional attributes those of OC. A distributed object of the class is transformed into a single-layer meta-object tower as follows. Let $MOatts(O) = (\mathtt{base} : O, \mathtt{up} : \mathtt{mt}, \mathtt{dwn} : \mathtt{mt}, \mathtt{in} : \mathtt{mt}, \mathtt{out} : \mathtt{mt})$ be the initial meta-object attributes for O. Then

$$o2m(\langle O : OC \mid atts \rangle) = \{ \ \langle \ O : o2m(OC) \mid atts, MOatts(O) \ \rangle \ \}.$$

The mapping is extended homomorphically to distributed object configurations (multi-sets of objects and messages) by defining $o2m$ to be the identity on messages.

For each rule of the object class there is a corresponding rule of the meta-object class obtained as follows. An object rule

$$r: \quad O \triangleleft M \ \ \langle O : OC \mid atts \rangle \longrightarrow \langle O : OC \mid atts' \rangle \ \ oCf$$

where oCf is a distributed object configuration, is mapped to the meta-object rule

$$o2m(r): \quad \langle O : o2m(OC) \mid atts, \mathtt{out} : L_o, \mathtt{in} : M \cdot L_i \ \rangle$$

$$\longrightarrow \langle O : o2m(OC) \mid atts', \mathtt{out} : L_o \cdot o2r(oCf), \mathtt{in} : L_i \rangle$$

where $o2r(oCf)$ is the request for creation of $o2m(oCf)$:

$$CreateConf(o2r(oCf)) = o2m(oCf).$$

Theorem 2.1: Given any distributed object class OC, in the union of the rewrite theories for the classes OC and $o2m(OC)$ the objects

$$\langle O : OC \mid atts \rangle \quad \text{and} \quad o2m(\langle O : OC \mid atts \rangle)$$

are observationally equivalent. More generally, let oCf be any distributed object configuration then oCf and $o2m(oCf)$ are observationally equivalent configurations.

To make sense of the above theorem we need to say what is meant by observational equivalence. Our approach is to show that the mapping $o2m$ can be extended to a mapping on computations (proof terms) in a manner that preserves communications. That is, we take the fundamental observation to be the identities of objects and the messages exchanged between them, ignoring the internal representation of state.

3 Composable Communication Services as Meta-Objects

By a communication *service* we mean a mechanism that ensures certain desirable properties about the communication between a family of distributed objects. For example, the communication medium may be faulty or insecure, and corresponding security or fault-tolerant services may be provided to ensure that messages do in fact arrive to their destinations uncorrupted, in the appropriate order, and without their information having been compromised because of malicious intruders. Similarly, performance reasons may dictate the convenience of using performance-boosting services, such as data compression, to achieve better quality of service with limited bandwith.

Our main proposal is to regard any such service as an appropriate class of meta-objects that, when installed in the desired family of distributed objects—for example, in the nodes of a particular network domain that is suspected to be insecure—mediate the communication between such objects ensuring certain guarantees. As explained in §2, the semantics of such a service is then the rewrite theory $\mathcal{S}$ formally specifying the behavior of meta-objects in the corresponding class.

What can we say about the *guarantees* that the use of a given service should offer? For services whose function is to protect the communication against possible faults and threats, so as to make it more survivable, such guarantees can be best explained in relationship to a formal specification of the type of *hostile environment* that the service is supposed to protect against. For example, in the case of a security service, this will require specifying formally the possible hostile behavior of a class of *intruders*. Similarly, for a fault-tolerant service, the guarantees should be made relative to a formal specification of the types of faults that the service can tolerate. In our approach, we can regard the formal specification of a hostile environment—in the context of which a service $\mathcal{S}$ can still provide some guarantees—as another rewrite theory $\mathcal{H}$.

The guarantees themselves are then formal properties that should be satisfied by a collection of objects in any distributed object class $\mathcal{A}$ when endowed with service $\mathcal{S}$ in the context of a hostile environment $\mathcal{H}$. That is, given an appropriate initial configuration of objects in $\mathcal{A}$ such that an $\mathcal{S}$-meta-object has been installed in each of the objects in the configuration, then the combined behavior of such a configuration—evolving according to the rewrite rules specified in $\mathcal{A}$ and $\mathcal{S}$—together with additional objects and rewrite rules specified by $\mathcal{H}$ satisfies certain formal properties. Furthermore, the guarantees should be independent of the object class $\mathcal{A}$ in question, provided $\mathcal{A}$ satisfies some minimal "interface requirements" making the service $\mathcal{S}$ suitable for $\mathcal{A}$. That is, the service $\mathcal{S}$ should be *parametric* in $\mathcal{A}$.

We have already explained in §2.3 that, under some simple assumptions, $\mathcal{A}$ can be transformed into a meta-object class. Adding the service $\mathcal{S}$ corresponds to considering configurations consisting of towers of height two with the $\mathcal{A}$-meta-objects at the bottom and the $\mathcal{S}$-meta-objects on top. Let us denote by $\mathcal{S}[\mathcal{A}]$ the rewrite theory governing the behavior of such height-two towers. Then, the formal properties ensured by service $\mathcal{S}$ in the context of a hostile environment $\mathcal{H}$ are properties of the combined rewrite theory $\mathcal{S}[\mathcal{A}] \cup \mathcal{H}$.

Therefore, such properties or guarantees are essentially *inductive properties* satisfied by the initial model [23] of the combined theory $\mathcal{S}[\mathcal{A}] \cup \mathcal{H}$. If P is a formula stating one of those guarantees, we use the notation

$$\mathcal{S}[\mathcal{A}] \cup \mathcal{H} \models_{ind} P[\mathcal{A}]$$

to express that $\mathcal{S}[\mathcal{A}] \cup \mathcal{H}$ satisfies the inductive property. Note the parametric nature of $P[\mathcal{A}]$, given that it depends on the actual parameter class $\mathcal{A}$. In general, P may depend on *two* parameters, namely, the stack $\mathcal{A}$ *underneath*, and the stack $\mathcal{L}$ of services *above*, service $\mathcal{S}$. Therefore, the most general notation for P is $\mathcal{L}[P[\mathcal{A}]]$. Of course, in this case $\mathcal{A}$ happens to be a height-one stack, and $\mathcal{L}$ is empty, and, therefore, it is not mentioned. The property $P[\mathcal{A}]$ can be expressed either in a first-order extension of rewriting logic, allowing for the usual logical connectives and quantification—as done in the examples discussed in §3.1 and §3.2—or in an appropriate temporal or modal logic.

This way of expressing service guarantees can help us deal with the problem of *composition* of services, and with the related problem of *combination* of the corresponding guarantees. That is, given services $\mathcal{S}$ and $\mathcal{T}$, respectively ensuring properties P and Q for hostile environments $\mathcal{H}$ and $\mathcal{G}$, what can we say about their composition? Do both properties hold in the combined hostile environment when both services are composed? And if so, in which order of composition? Indeed, given two services, we can consider two different compositions, namely $\mathcal{T}[\mathcal{S}[\mathcal{A}]]$ and $\mathcal{S}[\mathcal{T}[\mathcal{A}]]$. A very natural question to ask is when one of these compositions can provide the combined guarantees of both services, for example, say, when does

$$\mathcal{S}[\mathcal{T}[\mathcal{A}]] \cup \mathcal{H} \cup \mathcal{G} \models_{ind} P[\mathcal{T}[\mathcal{A}]] \wedge \mathcal{S}[Q[\mathcal{A}]]$$

hold? $P[\mathcal{T}[\mathcal{A}]]$ is the instantiation of property P for the actual parameter $\mathcal{T}[\mathcal{A}]$, and $\mathcal{S}[Q[\mathcal{A}]]$ is the instantiation of property Q with $\mathcal{A}$ underneath, and $\mathcal{S}$ above, service $\mathcal{T}$.

Overview of Case Study

In the remainder of this section we illustrate our approach to specification and reasoning about composable communication services with a simple case study. In §3.1 we specify a composable fault-tolerant service designed to provide reliable, in order message delivery in the presence of message loss and duplication faults The service is specified as a meta-object class `FTService` and the faults are specified via the object class `Saboteur`. In §3.2 we specify a composable security service designed to provide secure and authenticated message transmission in the presence of intruders that may observe, intercept, or fake messages. The service is specified as a meta-object class `EncrService` and the faults are specified via the object class `Intruder`. For each service we also state a formal service guarantee in the form discussed above. In §3.3 we discuss the composition of these services and their guarantees.

Our specifications are written in the executable rewriting logic language Maude [8]. The full specification is given in Appendix A, along with a brief introduction to the Maude syntax.

In order to simplify the presentation and focus on the main ideas we consider the special case of towers of height at most three. In particular, we will stack the security service `EncrService` on top of the fault-tolerant service `FTService`, which is stacked on top of an application `Application` (defined below). In this case, the fault-tolerant service will have a service above and the service below is the application. Thus, we consider properties of the fault-tolerant service of the form `1MOStack`[P[`Application`]] where `1MOStack` is a height-one meta-object stack, i.e., a meta-object class. Since the composition puts the encryption service on top of a fault-tolerant service, the secrecy property of the encryption service is Q[`1MOStack`[`Application`]]. To achieve the envisioned combination of the services, we instantiate `1MOStack` in the P-property with `FTService` and in the Q-property with `EncrService`.

We further simplify matters by considering the specific simple application class given below for the lowest level. An application agent can send and receive data. It has a list of data items stored in the `sndData` to be sent to another agent whose identity is stored in the `rec` attribute. Data received is stored in the `rcvData` attribute. The `Application` meta-object class has two rules, one for sending data and one for receiving.

```
class Application | rec : Oid, sndData : List[Data], rcvData : List[Data] .
```

```
subclass Application <  MetaObject .

var D : Data . var A B : Application . var DL : List[Data] .
var cL : List[MsgConf] . var L : List[Msg] .

rl [Send] :
< A : Application | out : cL, rec : B, sndData : D + DL >
=> < A : Application | out : (cL . (B <| D)), sndData : DL > .

rl [Receive] :
< B : Application | in : D . L, rcvData : DL >
=> < B : Application | in : L, rcvData : (DL + D)  > .
```

The service guarantee properties are formalized relative to an initial configuration involving both malicious objects and instances of meta-object towers which represent application objects with services installed on them. To make the ideas more concrete we consider initial configurations with two application objects `A` and `B` and services installed on top of them, and one malicious agent.

In order to guarantee properties of a service, either the malicious behavior has somehow to be limited—so that it is enough to reason on finite behaviors—or appropriate fairness assumptions are necessary. In this paper we take the first choice, and limit somewhat the injury capacity of malicious agents. Thus a first-order extension of rewriting logic can be used to formally state the service guarantees under consideration.

A more general treatment would consider any kind of instantiation of the system with well-behaved applications and malicious objects. In this case, formalizing service guarantees requires developing additional notation to express constraints such as 'application agents never compromise data to the malicious objects'. This would add substantially more detail, but the basic ideas would remain the same. Similarly, by moving to a temporal or modal logic to express fairness constraints, some of the explicit limitations on malicious agents could be removed.

3.1 A Composable Fault-Tolerant Service

Communication in a distributed system is not always reliable. Communication channels may be lossy or unreliable concerning the in-order delivery of messages. A fault-tolerant service is designed to overcome message loss and to maintain the order of messages. In the following we describe a fault-tolerant service specified in the executable rewriting logic language Maude [8]. Our specification creates a fault-tolerant service which guarantees a reliable, in-order communication mechanism in a distributed system. It specifies as a composable service a fault-tolerant protocol of the type described in [24, 16, 7].

A fault-tolerant meta-object (FT-meta-object) can process messages from different agents concurrently. Moreover, such a meta-object behaves both as a receiver and as a sender of messages. For this purpose, an FT-meta-object maintains two buffers: a send-buffer (`sndBuff`) and a receive-buffer (`rcvBuff`). In the send-buffer the FT-meta-object stores for each object the latest message counter, the currently processed message, and a waiting list of messages. In the receive-buffer the latest message counter for each agent is stored. The data items in the buffers are specified as follows.

```
sorts SndItem RcvItem .
op (_,_,_,_) : Oid MachineInt Default[Msg] List[Msg] -> SndItem .
op (_,_) : Oid MachineInt -> RcvItem .
```

The message counters are used to acknowledge and drop out-of-order messages. In the role of a sender, the FT-meta-object sends messages attached with a counter and the sender's name (`snd_#_from_`). In response to messages sent by another FT-meta-object, an acknowledgement with the latest message counter and the identity of the underlying base object is sent (`ack_from_`). The complete specification of the fault-tolerant service, including data types, can be found in Appendix A.

```
class FTService | sndBuff : Set[SndItem], rcvBuff : Set[RcvItem] .
subclass FTService < MetaObject .

msg snd_#_from_ :  Msg MachineInt Oid -> Msg .
msg ack_from_ : MachineInt Oid -> Msg .
```

The rules axiomatizing the behavior of an FT-meta-object are listed below. A message which is received by an FT-meta-object in its up-queue (i.e., from the next lower (meta-)object speaking in terms of a meta-object stack) is stored in the send-buffer. If it is the first message from an object, then a new entry in the send-buffer is created (`up/FirstMsg`). If the FT-meta-object does not currently process a message for an object, i.e., if the third position of the corresponding send-buffer item is `null`, then the FT-meta-object can take the next message from the waiting list (`Produce`). When sending out the current message, a message counter and the name of the base object are attached. A message which is received by a meta-object through its `in`-queue (i.e., from the next higher (meta-)object), is always acknowledged. If it is a new message, then it is also pushed down in the meta-object stack. The FT-meta-object only deletes a current message from its send-buffer if it has been acknowledged.

```
vars X A B : Oid . var L L' : List[Msg] . var cL : List[MsgConf],
vars I J : MachineInt . var M : Msg . var DMSG : Default[Msg] .
var SB : Set[SndItem] . var RB :  Set[RcvItem] .

rl [up] :
< X : FTService | up : (B <| M) . cL, sndBuff: SB + (B,I,DSMG,L) >
=> < X : FTService | up : cL, sndBuff : SB + (B,I,DMSG,(L . M)) > .

crl [up/FirstMsg] :
< X : FTService | up : (B <| M) . cL, sndBuff: SB >
=> < X : FTService | up : cL, sndBuff : SB + (B,0,null,M) >
if not(B in projOids(SB))  .

rl [Produce] :
< X : FTService | sndBuff: SB + (B,I,null,(M.L)) >
=> < X : FTService | sndBuff : SB + (B,I+1,M,L) > .

rl [out/Send] :
< X : FTService | sndBuff : SB + (B,I,M,L), out : cL, base : A >
=> < X : FTService | out : cL . (B <| snd(M)#(I)from(A)) > .

rl [in/Ack] :
< X : FTService | in : (ack(J)from(B)) . L, sndBuff : SB + (B,I,M,L') >
=> if (I == J)
   then < X : FTService | in : L, sndBuff : SB + (B,I,null,L') >
   else < X : FTService | in : L > fi .

rl [in/Msg,out/Ack] :
< X : FTService | in : snd(M)#(I)from(A) . L, out : cL, dwn : L',
                  base : B, rcvBuff : (RB u (A,J))  >
=> if (I == J + 1)
   then < X : FTService | in : L, out : cL . (A <| ack(I)from(B)),
                          dwn : L' . M, rcvBuff : RB u (A,I) >
   else < X : FTService | in : L, out : cL . (A <| ack(I)from(B)) > fi .

rl [in/FirstMsg,out/Ack] :
< X : FTService | in : snd(M)#(1)from(A) . L,
                  out : cL, dwn : L', base : B, rcvBuff : RB >
=> < X : FTService | in : L, out : cL . (A <| ack(1)from(B)),
                     dwn : L' . M, rcvBuff : RB u (A,1) > .
```

A Faulty Environment

A communication medium can be faulty, i.e., it may not provide reliable, in-order delivery of messages. We can specify such a hostile environment by postulating sabotaging agents that may duplicate or destroy messages. To ensure that messages are not always destroyed, we limit the injury capacity of a saboteur.

```
(omod SABOTEUR is
  class Saboteur | cnt : MachineInt .

  var A S : Oid . var M : Msg . var I : MachineInt .
```

```
  rl [Duplicate] :
  (A <| M)   < S : Saboteur | cnt : I >
  => (A <| M) (A <| M) < S : Saboteur | cnt : I > .

  crl [Destroy] :
  (A <| M)  < S : Saboteur | cnt : I >
  =>  < S : Saboteur | cnt : I - 1 > if N > 0 .
endom)
```

Fault-Tolerant Service Guarantee

The fault-tolerant service should be immune to message duplication and to limited deletion of messages. It should guarantee reliable, in-order arrival of a list of messages in spite of the above faulty environment. The property is stated for a specific class of configurations which satisfy the following pattern for their initial state:

```
eq initP(A,AFT,B,BFT,D,L,RD,S,N,MO,MO') =
   { MO o < AFT : FTService | sndBuff : mt, rcvBuff : mt, base : A > o
          < A : Application | rec : B, sndData : D + L > }
   { MO' o < BFT : FTService | sndBuff : mt, rcvBuff : mt, base : B > o
           < B : Application | rcvData : RD > }
   < S : Saboteur | cnt : N >  .
```

for `D: Data, L, RD: List[Data], N: MachineInt, MO, MO': 1MOStack`.

We want to guarantee that the fault-tolerance service provides a reliable service on top of an unreliable medium in which a sabotaging agent is present. That is, in any computation one can reach a state in which the messages have been delivered in correct order to the receiver object. This can be stated as follows:

$$\forall \texttt{A,AFT,B,BFT,S,D,L,RD,N,MO,MO'}, \forall \texttt{C: Configuration} \bullet$$
$$(\texttt{A} \neq \texttt{B} \wedge \texttt{initP(A,AFT,B,BFT,D,L,RD,S,N,MO,MO')} \longrightarrow \texttt{C})$$
$$\Rightarrow \exists \texttt{C': Configuration} \bullet (\texttt{C} \longrightarrow \texttt{C'}\ \{\ \texttt{< B : Application | rcvData : RD + D + L>}\ \})$$

In order to prove this property some additional constraints about the interfaces of the service meta-objects `MO` and `MO'` are necessary. Only if the intermediate services behave properly can the reliability property be shown. In order to prove this one can instantiate the property for a specific service stack, as will be done in §3.3.

3.2 A Composable Encryption Service

There is a variety of security protocols in the literature. Protocols can be divided into those which use an asymmetric cryptosystem and protocols which uses a symmetric cryptosystem. In this paper we present an encryption service that uses asymmetric keys. In an asymmetric cryptosystem different keys are used for encryption and decryption. The decryption keys must be kept secret, whereas the encryption keys are publicly available. For this reason, asymmetric key encryption is also often called *public-key encryption*. An appropriate cryptographic data type specification for public-key encryption is as follows. The data being encrypted is called a “field” in cryptographic literature.

```
(fmod CRYPTO-DT is
  protecting BOOL .
  protecting CONFIGURATION .
  sort Field PKey .
  subsort PKey < Field .
  op _,_ : Field Field -> Field [assoc] .
  op {_}_ : Field PKey -> Field .                  *** de/encryption
  op keypair : PKey PKey -> Bool [comm] .
```

```
  op pk_ : Oid -> PKey .                          *** public key
  op sk_ : Oid -> PKey .                          *** secret key
  var O : Oid . var F : Field . vars SK PK : Pkey .
  eq keypair(sk(O),pk(O)) = true .
  ceq {{F}PK}SK = F if keypair(PK,SK) .
 endfm)
```

An encryption service has to take care that each message which is passed from a lower level is encrypted with the public key of the addressee. Messages which are received from an upper level are either decrypted with the secret key of the base object, or discarded if the encryption key and the secret key of the base object are not a keypair. Encryption will prevent an intruder from seeing the contents of a message (secrecy), but it does not guarantee authenticity. This means that an intruder can fool legal agents of the protocol by faking an encrypted message with the public key of another agent. Thus, in order to provide a service which guarantees security and authenticity, we need to sign messages. *Sealing* is a common technique in cryptography to sign messages. A message is encrypted with the secret key of an agent (`seal`). A signed message can be verified (`verify`). We extend the cryptographic data type to include a sealing mechanism.

```
(fmod CRYPTO-DT is
  ...
  sort Pseal .
  subsort Pseal < Field .
  op seal : PKey Field -> Pseal .
  op verify : PKey PSeal Field -> Bool .
  vars PK SK : PKey, var F : Field .
  ceq verify(PK, seal(SK, F), F) = true if keypair(PK,SK) .
 endfm)
```

The encryption service can use the sealing mechanism in the following way. If a field is passed from a lower-level meta-object in the stack, then the encryption service can sign the field. This signature is composed with the field and the name of the sender object, and all three items are encrypted with the public key of the receiver. Correspondingly, for incoming messages, first the message is decrypted, and for the resulting triple (sender identity, message, signature) the signature is verified. If this test succeeds, then the message is forwarded downward to the next lower level in the stack.

```
(omod ENCR is
  protecting METAOBJECT .
  protecting CRYPTO-DT .

  class EncrService .
  subclass EncrService < MetaObject .

  vars X A B : Oid . var PK SK : PKey . var  cL cL' : List[MsgConf] .
  var M : Msg . var S : PSeal .

  rl [up/Encryption] :
  < X : EncrService | up : (B <| M) . cL, out : cL', base : A >
  => < X : EncrService | up : cL, out : cL' . (B <| {A,M,seal(sk(A),M)}pk(B)) > .

  rl [in/Decryption] :
  < X : EncrService | in : ({B,M,S}PK) . cL, dwn : cL', base : A >
  => if (keypair(sk(A),PK) and verify(pk(B),S,M))
     then < X : EncrService | in : cL, dwn : cL' . M >
     else < X : EncrService | in : cL > fi .
 endom)
```

An Intruder Model

Dolev and Yao [9] proposed the following behavior of an intruder for security protocols: An intruder may observe, intercept or fake messages. Introducing a fake message into the system either means replaying a previously observed or intercepted message, or building a new message from the data which is available to the intruder. For the latter purpose, the intruder can decompose and compose message fields of seen messages, and the intruder may encrypt or decrypt messages with keys he possesses. We model the knowledge of an intruder as one particular attribute, in which every overheard message is stored. The content of this attribute is then the basis for further analysis by the intruder, such as message (de)composition, de/encryption. The intruder can overhear messages and has a limited capacity of faking and intercepting messages. A complete specification of the class `Intruder` is given in Appendix A.

Encryption Service Guarantee

"Secrecy" is obviously a goal for an encryption communication service. Secrecy means, that a spy cannot read the contents of a message. But authenticity is also important, i.e., if a message appears to be from a specific agent, then this agent did send the message. We define the secrecy property for a system configuration with three agents, two application objects `A` and `B` and one intruder `I`. We assume that `A` has some data `D` to send to agent `B`. The initial configuration has to satisfy the following general pattern:

```
eq initQ(A,AE,B,BE,D,O,SD,RD,I,K,MO,MO') =
   { < AE : EncrService | buff : mt, base: A > o
     MO o < A : Application | rec : B, sndData : D > }
   { < BE : EncrService | buff : mt, base: B > o
     MO' o < B : Application | sndData : SD, rcvData : RD, rec : O > }
   < I : Intruder | knows : K >,
```

where `D: Data, SD, RD: List[Data], K: Set[Item], MO, MO': 1MOStack`. The sort `OidDefault` is a supersort of `Oid` which provides the default identifier `none` (`O: OidDefault`).

Secrecy is expressed by the following formula:

$$\forall \texttt{A,AE,B,BE,D,O,SD,RD,I,K,MO,MO'}, \forall \texttt{C: Configuration, K': Set[Item]} \bullet$$
$$(\texttt{A} \neq \texttt{B} \wedge \texttt{D} \notin \texttt{K} \wedge (\texttt{D} \notin \texttt{SD} \vee \texttt{O} \neq \texttt{I}) \wedge$$
$$\texttt{initQ(A,AE,B,BE,D,O,SD,RD,I,K,MO,MO')} \rightarrow \texttt{C <I : Intruder | knows : K'>})$$
$$\Rightarrow \texttt{D} \notin \texttt{K'}.$$

This secrecy assertion guarantees that, for an initial configuration with two honest parties and one intruder, where one of the honest agents has a data item which is initially not known to the intruder and not to the other agent ($\texttt{D} \notin \texttt{SD}$) (or, if the other agent knows this data, he will not communicate it to the intruder ($\texttt{O} \neq \texttt{I}$)), then the intruder can never learn the data. $\texttt{C} \rightarrow \texttt{C'}$ means that there is a computation between the corresponding configurations.

3.3 Composing Encryption and Fault-Tolerant Services

The communication services specified in the previous sections can be stacked on top of each other with an application at the bottom in order to create a secure, reliable communication service. The interesting question is what composition achieves the desired combination of properties. For example, stacking the fault-tolerance service on top of the encryption service would not work. The reason for this is that the fault-tolerance service can overcome a lossy medium, but it is not designed to overcome a medium that corrupts or fakes messages. In a composition where the fault-tolerant service is on top of the secure communication service, the number flag from the fault-tolerant service is not protected, and thus, could be changed by a hostile environment or an acknowledgement could be faked. The guarantee of the fault-tolerance service, i.e., in-order guaranteed arrival of messages, would no longer hold true. However, a secure and reliable communication service can be achieved by the alternative composition: `ENCRService[FTService[Application]]`. An example meta-object tower has then the following form:

```
{ < AE : Encryption | base : A > o
  < AFT : FTService | sndBuff : mt, rcvBuff : mt > o
  < A : Application | rec : B, sndData : D >  }
```

Properties of the Composition

The properties of the composed system are derived by instantiating the free service parameter in the properties of each service. Given the service stack `ENCRService[FTService[Application]]` we get the following conjunction of properties:

$$\mathtt{E}[P[\mathtt{A}]] \wedge Q[\mathtt{FT}[\mathtt{A}]] := \\ \mathtt{EncrService}[P[\mathtt{Application}]] \wedge Q[\mathtt{FTService}[\mathtt{Application}]]$$

where $\mathtt{E}[P[\mathtt{A}]]$ is the reliability property, and $Q[\mathtt{FT}[\mathtt{A}]]$ is the secrecy property.

The desired combined property of the composition of servives can then be expressed as

$$\mathtt{EncrService[FTService[Application]]} \cup \mathtt{Saboteur} \cup \mathtt{Intruder} \\ \models_{ind} \mathtt{E}[P[\mathtt{A}]] \wedge Q[\mathtt{FT}[\mathtt{A}]].$$

4 Conclusions and Future Work

We have given a precise rewriting logic semantics to an important reflective mechanism that permits composing different distributed services in a modular and principled way. As a fruit of this semantics we have also shown how the formal properties expressing different service guarantees—relative to different hostile environments—can themselves also be stated in a parametric way, so that one can combine in a systematic way the properties of different services when composing them, in order to study whether they hold or not for a given composition of services.

We have illustrated the ideas by means of a fault tolerant service and an encryption service, showing how they can be formalized as services in our sense, and we have also specified typical hostile environments for these services and typical guarantees that one would like to ensure. We then have discussed the composition of these two services and the combined guarantees that one would like to establish.

We view this work as a first step in an effort to apply the methods of rewriting logic and distributed object-oriented reflection to the area of composable distributed services. Much more work remains ahead, particularly in the following aspects:

- For the sake of simplicity, we have used a relatively simple formalism for stating service guarantees, namely, first-order rewriting logic. A number of useful properties can be stated this way, but in general one would like to state properties about *infinite* behaviors that may depend on assumptions such as fairness. Similarly, for some services real-time properties may be quite important, and one would then like to use a formalism supporting real-time aspects. We plan to study a variety of formalisms—including different temporal and modal logics—for expressing these richer properties.
- Together with an investigation of adequate property-oriented formalisms to express service guarantees, we plan to develop associated proof methods, including inductive proof techniques, and both deductive and model-checking approaches to proving temporal logic properties, and taking into account work such as [28, 10, 7].
- Different meta-object composition mechanisms should be investigated. Besides linear towers one should consider other structures such as trees. Even within linear towers, mechanisms for localizing the services to specific groups of objects should be investigated, so that messages not related to such objects can bypass the service in question.
- More examples and case studies are neeeded to advance these ideas. Besides exploring different composition mechanisms, we should also study services with different modes of communication. For example, besides point-to-point communication we should investigate broadcast and multicast services.
- The topic of *interfaces* for services and their semantic properties is also of outmost importance and should be further investigated. We expect that the parameterized specification techniques supported by rewriting logic and Maude [8] will be particularly useful in this regard.

- An important topic not treated in this paper is *dynamic installation* of services. Developing an appropriate semantics for the reflective mechanims of such installations seems a very worthwhile topic of research.
- Optimizing meta-object towers in a semantics-preserving way is also important. We expect that theory transformation techniques will be very useful in this regard (cf. [32]).
- Our meta-object tower of reflection differs in some details from that proposed by Agha, Astley, and Sturman [33, 4]. The main difference is the level of synchrony between an object and its meta-object. A study of the formal relation between the different models is a interesting topic for future work.

References

[1] M. Abadi and L. Lamport. Conjoining specifications. Technical Report 118, DEC Systems Research Center, 1994.

[2] G. Agha. Abstracting interaction patterns: A programming paradigm for open distributed systems. In E. Najm and J.-B. Stefani, editors, *Formal Methods for Open Object-based Distributed Systems*, pages 135–153. Chapman & Hall, 1997.

[3] G. Agha, S. Frølund, W. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and Modularity Mechanisms for Concurrent Computing. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(2):3–14, May 1993.

[4] M. Astley and G. Agha. Customization and composition of distributed objects: Middleware abstractions for policy management. In *Sixth International Symposium on the Foundations of Software Engineering (FSE-6/SIGSOFT'98)*, 1998.

[5] H. G. Baker and C. Hewitt. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992. IFIP, Aug. 1977.

[6] K. M. Chandy and J. Misra. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[7] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[8] Clavel, M. and Durán, F. and Eker, S. and Lincoln, P. and Martí-Oliet, N. and Meseguer, J. and Quesada, J. *Maude: Specification and Programming in Rewriting Logic*. SRI International, Computer Science Laboratory, Menlo Park, CA, January 1999. URL: `http://maude.csl.sri.com/manual/`.

[9] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29:198–208, 1983. Also STAN-CS-81-854, May 1981, Stanford U.

[10] B. Dutertre and S. Schneider. Using a PVS Embedding of CSP to Verify Authentication Protocols. In *Theorem Proving in Higher Order Logics, TPHOL's 97*, pages 121–136. Springer, August 1997. LNCS 1275.

[11] Feldmeier, D. and McAuley, A. and Smith, J. and Bakin, D. and Marcus, W. and Raleigh, T., 1999. Accepted for IEEE JSAC Special Issue on "Protocol Architecures for the 21st Century." Available at URL: `http://carin.bellcore.com:8000/boosters/` under papers.

[12] S. Frølund. *Coordinated Distributed Objects: An Actor Based Approach to Synchronization*. MIT Press, 1996.

[13] S. Frølund and G. Agha. A language framework for multi-object coordination. In *Proceedings of ECOOP 1993*, volume 707 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

[14] C. B. Jones. Specification and design of parallel programs. In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 321–332, 1983.

[15] Kiczales, G. and Lamping, J. and Mendhekar, A. and Maeda, C. and Lopes, C. and Loingtier, J.-M. and Irwin, J. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proc. of the European Conference on Object-Oriented Programming (ECOOP'97), Finland*, pages 220–242, 1997. LNCS 1241.

[16] S. S. Lam and A. U. Shankar. A relational notation for state transition systems. *Software Engineering*, 16(7):755–775, July 1990.

[17] T. F. LaPorta, D. Lee, Y.-J. Lin, and M. Yannakakis. Protocol Feature Interactions. In S. Budkowski, A. Cavalli, and E. Najm, editors, *Proc. Formal Description Techniques And Protocol Specification, Testing and Verification, FORTE XI/PSTV XVIII'98, 3-6 November, Paris, France*, pages 59–74, 1998.

[18] P. Lincoln, N. Martí-Oliet, and J. Meseguer. Specification, transformation, and programming of concurrent systems in rewriting logic. In G. Blelloch, K. Chandy, and S. Jagannathan, editors, *Specification of Parallel Algorithms*, pages 309–339. DIMACS Series, Vol. 18, American Mathematical Society, 1994.

[19] A. Mallet, J. D. Chung, and J. M. Smith. Operating System Support for Protocol Boosters, 1998. available at URL: `http://carin.bellcore.com:8000/boosters/` under papers.

[20] I. A. Mason and C. L. Talcott. A semantically sound actor translation. In *ICALP'97*, number 1256 in Lecture Notes in Computer Science, pages 369–378, 1997.

[21] I. A. Mason and C. L. Talcott. Actor languages: Their syntax, semantics, translation, and equivalence, 1999. To appear.

[22] J. Meseguer. A logical theory of concurrent objects. In *ECOOP-OOPSLA'90 Conference on Object-Oriented Programming, Ottawa, Canada, October 1990*, pages 101–115. ACM, 1990.

[23] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[24] J. Meseguer. A Logical Theory of Concurrent Objects and Its Realization in the Maude Language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. The MIT Press, 1993.

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

[26] J. Meseguer and C. Talcott. A Partial Order Event Model for Concurrent Objects, 1998. submitted for publication.

[27] J. Millen. Local Reconfiguration Policies. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1999. *To appear*.

[28] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1):85–128, 1998.

[29] J. Peleska. On a unified formal approach for the development of fault-tolerant and secure systems. In H. Rischel, editor, *Nordic Seminar on Dependable Computing Systems, Lyngby, Denmark, August 1994. Technical University of Denmark*, pages 69–80, 1994.

[30] S. Ren. *An Actor-Based Framework for Real-Time Coordination*. PhD thesis, University of Illinois at Champaign Urbana, 1997.

[31] J. Rushby. Combining system properties: A cautionary example and formal examination. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, June 1995. Unpublished project report; available at URL: `http://www.csl.sri.com/ rushby/combined.html`.

[32] S. F. Smith and C. L. Talcott. Modular reasoning for actor specification diagrams, 1999. (to appear in FMOODS99).

[33] D. Sturman. *Modular Specification of Interaction Policies in Distributed Computing*. PhD thesis, University of Illinois at Champaign Urbana, 1996.

[34] D. Sturman and G. Agha. A protocol description language for customizing failure semantics. In *The 13th Symposium on Reliable Distributed Systems, Dana Point, California*. IEEE, Oct. 1994.

[35] C. L. Talcott. An actor rewriting theory. In J. Meseguer, editor, *Proc. 1st Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 360–383. North Holland, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.

[36] C. L. Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3), 1998.

A Complete Maude Specifications

This Appendix presents the complete Maude specifications of the data types, classes, and meta-object classes discussed in this paper.

We briefly summarize the syntax of Maude. There are three types of modules:

- *functional* modules, that are equational theories used to specify algebraic data types; they are declared with the syntax `fmod ... endfm`,
- *system* modules, that are rewrite theories specifying concurrent systems; they are declared with the syntax `mod ... endm`, and
- *object-oriented* modules, that provide special syntax to specify concurrent object-oriented systems but are entirely reducible to system modules; they are declared with the syntax `omod ... endom`.

All the above modules can be *parameterized*, and have an *initial model semantics*. That is, they specify the initial models of their corresponding theories, or, in the case of parameterized modules, the free extension of a model of the parameter theory to a model of the body.

Immediately after the module's keyword, the *name* of the module is given. After this, a list of imported submodules can be added. One can also declare *sorts*—that is, types—and *subsorts*—that is, subtype inclusions. In object-oriented modules one can declare *classes* and *subclasses*. Each class is declared with the syntax

$$\texttt{class}\ C \mid a_1 \texttt{:}\ S_1\texttt{,}\ \ldots\texttt{,}\ a_n \texttt{:}\ S_n$$

where C is the name of the class, and for each $a_i : S_i$, a_i is an attribute identifier, and S_i is the sort inside which the values of such an attribute identifier must range.

One can also declare *operators*, introduced with the `op` keyword. They can have user-definable syntax, with underbars '_' marking the argument positions, and are declared with the sorts of their arguments and the sort of their result. Some operators can have equational *attributes*, such as `assoc` and `comm`, stating, for example, that the operator is associative and commutative. Such attributes are then used by the Maude engine to match terms *modulo* the declared axioms. In object-oriented modules, message-constructing operators are introduced with the keyword `msg`.

There are three kinds of logical axioms, namely, *equations*—introduced with the keywords `eq`, or, for conditional equations, `ceq`—*memberships*—declaring that a term has a certain sort and introduce with the keywords `mb` and `cmb`—and *rewrite rules*—introduced with the keywords `rl` and `crl`. Functional modules can only have equations and memberships. System and object-oriented modules can have any of the three kinds of axioms. The mathematical variables in such axioms are declared with the keywords `var` and `vars`.

The parameters of parameterized modules are *theories*. Theories are entirely analogous to modules, except that they have a *loose semantics*, that is, any model satisfying the axioms of the theory is acceptable. Theories can be functional, system, or object-oriented, and are declared with respective syntax `fth ... endfth`, `th ... endth`, and `oth ... endoth`. A parameterized module `M` with parameter theories, say, `T1, ..., Tn` is written in the form `M[X1 :: T1, ..., Xn :: Tn]`.

The instantiation of a parameterized module is accomplished by *views*. Views are mappings binding the sorts and operators of parameter theories to their corresponding instantiations in the module chosen as the actual parameter. Views are declared with the syntax `view ... endv`. View names can then be placed in the corresponding parameters of a parameterized module to instantiate the module.

Basic Data Types

```
(fth TRIV is
  sort Elt .
 endfth)

(fmod LIST[X :: TRIV] is
  protecting MACHINE-INT .
  sort List[X] .
  subsort Elt.X < List[X] .
  op nil : -> List[X] .
  op __ : List[X] List[X] -> List[X] [assoc id: nil] .
  op length : List[X] -> MachineInt .
  var E : Elt.X . var L : List[X] .
  eq length(nil) = 0 .
  eq length((E L)) = length(L) + 1 .
 endfm)

(fmod SET[X :: TRIV] is
  protecting BOOL .
  sorts Set[X] NeSet[X] .
  subsorts Elt.X < NeSet[X] < Set[X] .
  op mt : -> Set[X] .
  op __ : Set[X] Set[X] -> Set[X] [assoc comm id: mt] .
  op _\_ : Set[X] X -> Set[X] .
  op _in_ : Elt.X Set[X] -> Bool .
  vars E E' : Elt.X . var S : Set[X] .
  eq E E = E .
  eq E in mt = false .
  eq E in (E S) = true .
  ceq E in (E' S) = E in S if not (E == E') .
  eq S \ E = S not(E in S) .
  eq (E S) \ E = S .
 endfm)

(fth FUNCTION is
  sorts Domain Codomain .
  op f : Domain -> Codomain .
 endfth)

(view Domain from TRIV to FUNCTION is
  sort Elt to Domain .
 endv)

(view Codomain from TRIV to FUNCTION is
  sort Elt to Codomain .
 endv)

(fmod MAP[F :: FUNCTION] is
  protecting (SET[Domain] * (op __ to _;_))[F] .
  protecting SET[Codomain][F] .
  op map : Set[Domain][F] -> Set[Codomain][F] .
  var A : Domain.F .
  var S : Set[Domain][F] .
  eq map(mt) = (mt).Set`[Codomain`]`[F`] .
  eq map(A ; S) = f(A) map(S) .
 endfm)
```

Meta-Objects

```
(omod MESSAGECONFIGURATION is
  sort MsgConf .
  subsort Msg < MsgConf < Configuration .
  op __ : MsgConf MsgConf -> MsgConf [assoc comm id: empty] .
 endom)

(view MsgConf from TRIV to MESSAGECONFIGURATION is
  sort Elt to MsgConf .
 endv)

(view Msg from TRIV to CONFIGURATION is
  sort Elt to Msg .
 endv)

(omod MESSAGEPACKET is
  op _<|_ : Msg Oid -> Msg .
 endom)


(omod METAOBJECT is
  protecting LIST * (op __ to _._) [MsgConf] .
  protecting LIST * (op __ to _;_) [Msg] .
  protecting MESSAGEPACKET .

  class MetaObject | in : List[Msg], out : List[MsgConf],
                     up : List[MsgConf], dwn : List[Msg], base : Oid .

  sort MOStack .
  subsort MetaObject < MOStack .

  sort MOTower .
  subsort MOTower < Configuration .

  op mtStack : -> MOStack .
  op _o_ : MOStack MOStack -> MOStack [assoc id: mtStack] .
  op {_} : MOStack -> MOTower .
  op createConf : MsgConf -> Configuration .

  vars A X Y : Oid . var M : Msg . vars cL cl' : List[MsgConf] . vars L L' : List[Msg] .
  var M : Msg . var MCONF : MsgConf . var ATT : AttributeSet . vars MOS : MOStack .

  rl [In] :
  (A <| M)
  { < X : MetaObject | in : L, base : A > o MOS }
  => { < X : MetaObject | in : L ; M > o MOS }  .

  rl [Out] :
  { < X : MetaObject | out : MCONF . cL  > o MOS }
  => { < X : MetaObject | out : cL > o MOS } createConf(MCONF) .

  rl [Down] :
  < X : MetaObject | dwn : MCONF . L' > o < Y : MetaObject | in : L >
  =>
  < X : MetaObject | dwn : L' > o < Y : MetaObject | in : L . MCONF >

  rl [Up] :
  < X : MetaObject | up : cL' > o < Y : MetaObject | out : MCONF . cL >
  =>
  < X : MetaObject | up : cL' . MCONF > o < Y : MetaObject | out : cL >
 endom)
```

Application

```
(fth DATA is
```

```
  sort Data .
 endfth)

(view Data from TRIV to DATA is
  sort Elt to Data .
 endv)

(omod APPLICATION is
   protecting METAOBJECT .
   protecting LIST * (op __ to _+_) [Data] .
   subsort Data < Msg .

   class Application | rec : Oid, sndData : List[Data], rcvData : List[Data] .
   subclass Application <  MetaObject .

   var D : Data . vars A B : ApplAgent . var DL : List[Data] .
   var cL : List[MsgConf] . var L : List[Msg] .

   rl [Send] :
   < A : Application | out : cL, rec : B, sndData : D + DL >
   => < A : Application | out : (cL . (B <| D)), sndData : DL > .

   rl [Receive] :
   < B : Application | in : D . L, rcvData : DL >
   => < B : Application | in : L, rcvData : (DL + D)  > .
 endom)
```

Fault-Tolerant Service

```
(fmod DEFAULT[Y :: TRIV] is
  sort Default[Y] .
  subsort Elt.Y < Default[Y] .
  op null : -> Default[Y] .
 endfm)

(fmod BUFFITEM is
  protecting CONFIGURATION .
  protecting MACHINE-INT .
  protecting DEFAULT[Msg] .
  protecting LIST * (op __ to _._) [Msg] .

  sorts SndItem RcvItem .
  op (_,_,_,_) : Oid MachineInt Default[Msg] List[Msg] -> SndItem .
  op (_,_) : Oid MachineInt -> RcvItem .
  op _@obj : SndItem -> Oid .

  var O : Oid . var M : MachineInt . var D : Default[Msg] . var L : List[Msg] .

  eq (O,M,D,L)@obj = O .
 endfm)

(view ProjOid from FUNCTION to BUFFITEM
  sort Domain to SndItem .
  sort Codomain to Oid .
  op f to @obj .
 endv)

(omod FT is
  protecting METAOBJECT .
  protecting DEFAULT[Msg] .
  protecting BUFFITEM .
  protecting MAP  [ProjOid] * (op map to projOids) .
  protecting SET * (op __ to +) [SndItem] .
  protecting SET * (op __ to u) [RcvItem] .
```

```
  class FTService | sndBuff : Set[SndItem], rcvBuff : Set[RcvItem] .
  subclass FTService < MetaObject .

  msg snd_#_from_ :  Msg MachineInt Oid -> Msg .
  msg ack_from_ : MachineInt Oid -> Msg .

  vars X A B : Oid . vars L L' : List[Msg] . var cL : List[MsgConf],
  vars I J : MachineInt . var M : Msg . var DMSG : Default[Msg] .
  var SB : Set[SndItem] . var RB :  Set[RcvItem] .

  rl [up] :
  < X : FTService | up : (B <| M) . cL, sndBuff: SB + (B,I,DSMG,L) >
  => < X : FTService | up : cL, sndBuff : SB + (B,I,DMSG,(L . M)) > .

  crl [up/FirstMsg] :
  < X : FTService | up : (B <| M) . cL, sndBuff: SB >
  => < X : FTService | up : cL, sndBuff : SB + (B,0,null,M) >
  if not(B in projOids(SB))  .

  rl [Produce] :
  < X : FTService | sndBuff: SB + (B,I,null,(M.L)) >
  => < X : FTService | sndBuff : SB + (B,I+1,M,L) > .

  rl [out/Send] :
  < X : FTService | sndBuff : SB + (B,I,M,L), out : cL, base : A >
  => < X : FTService | out : cL . (B <| snd(M)#(I)from(A)) > .

  rl [in/Ack] :
  < X : FTService | in : (ack(J)from(B)) . L, sndBuff : SB + (B,I,M,L') >
  => if (I == J)
     then < X : FTService | in : L, sndBuff : SB + (B,I,null,L') >
     else < X : FTService | in : L > fi .

  rl [in/Msg,out/Ack] :
  < X : FTService | in : snd(M)#(I)from(A) . L, out : cL, dwn : L',
                    base : B, rcvBuff : (RB u (A,J))  >
  => if (I == J + 1)
     then < X : FTService | in : L, out : cL . (A <| ack(I)from(B)),
                            dwn : L' . M, rcvBuff : RB u (A,I) >
     else < X : FTService | in : L, out : cL . (A <| ack(I)from(B)) > fi .

  rl [in/FirstMsg,out/Ack] :
  < X : FTService | in : snd(M)#(1)from(A) . L,
                    out : cL, dwn : L', base : B, rcvBuff : RB >
  => < X : FTService | in : L, out : cL . (A <| ack(1)from(B)),
                       dwn : L' . M, rcvBuff : RB u (A,1) > .
 endom)
```

Faulty Environment

```
(omod SABOTEUR is
  class Saboteur | cnt : MachineInt .

  vars A S : Oid . var M : Msg . var I : MachineInt .

  rl [Duplicate] :
  (A <| M)   < S : Saboteur | cnt : I >
  => (A <| M) (A <| M) < S : Saboteur | cnt : I > .

  crl [Destroy] :
  (A <| M)  < S : Saboteur | cnt : I >
  =>  < S : Saboteur | cnt : I - 1 > if N > 0 .
endom)
```

Security Service

```
(fmod CRYPTO-DT is
  protecting BOOL .
  protecting CONFIGURATION .
  sort Field PKey Pseal .
  subsorts Pseal PKey < Field .

  op _,_ : Field Field -> Field [assoc] .
  op {_}_ : Field PKey -> Field .            *** de/encryption
  op keypair : PKey PKey -> Bool [comm] .
  op pk_ : Oid -> PKey .                     *** public key
  op sk_ : Oid -> PKey .                     *** secret key
  op seal : PKey Field -> Pseal .
  op verify : PKey PSeal Field -> Bool .

  var O : Oid .  vars PK SK : PKey, var F : Field .
  eq keypair(sk(A),pk(A)) = true .
  ceq {{F}PK}SK = F if keypair(PK,SK) .
  ceq verify(PK, seal(SK, F), F) = true if keypair(PK,SK) .
 endfm)

(omod ENCR is
  protecting METAOBJECT .
  protecting CRYPTO-DT .

  class EncrService .
  subclass EncrService < MetaObject .

  vars X A B : Oid . vars PK SK : PKey . vars cL cL' : List[MsgConf] .
  var M : Msg . var S : PSeal .

  rl [up/Encryption] :
  < X : EncrService | up : (B <| M) . cL, out : cL', base : A >
  => < X : EncrService | up : cL, out : cL' . (B <| {A,M,{M}sk(B)}pk(B)) > .

  rl [in/Decryption] :
  < X : EncrService | in : ({B,M,S}PK) . cL, dwn : cL', base : A >
  => if (keypair(sk(A),PK) and verify(pk(B),S,M))
     then < X : EncrService | in : cL, dwn : cL' . M >
     else < X : EncrService | in : cL > fi .
 endom)
```

Intruder – Security Service

```
(fmod ITEM is
  protecting CRYPTO-DT .
  protecting CONFIGURATION .
  sort Item .
  subsort Msg < Item .
  subsort Field < Item .
  subsort Oid < Item .
 endfm)

(view Item from TRIV to ITEM
  sort Elt to Item .
 endv)

(omod INTRUDER is
  protecting METAOBJECT .
  protecting CRYPTO-DT .
  protecting SET * ( op __ to _+_) [Item] .

  class Intruder | knows : Set[Item], cnt : MachineInt .

  vars A X : Oid . var M : Msg . var H : Set[Item] . vars K' K : PKey .
```

```
 vars F F' : Field . var I : MachineInt . vars N N' : Nonce .

 rl [OverhearMessage] :
 (A <| M) < X : Intruder | knows : H >
 => (A <| M)  < X : Intruder | knows : H + (A <| M) > .

 crl [InterceptMessage] :
 (A <| M) < X : Intruder | knows : H, cnt : I >
 => < X : Intruder | knows : H + (A <| M), cnt : I -1 >
 if (I > 0) .

 crl [FakeMessage] :
 < X : Intruder | knows : H + (A <| M), cnt : I  >
 => < X : Intruder | knows : H + (A <| M), cnt : I - 1 > (A <| M)
 if (I > 0) .

 crl [Decryption] :
 < X : Intruder | knows : H + K + {F}K' >
 => < X : Intruder | knows : H + K + {F}K' + F >
 if keypair(K,K') .

 rl [Encryption] :
 < X : Intruder | knows : H + X + K >
 => < X : Intruder | knows : H + X + K + {X}K > .

 rl [Decomposition1] :
 < X : Intruder | knows : H + (A <| M) >
 => < X : Intruder | knows : H + (A <| M) + A + M > .

 rl [Decomposition2] :
 < X : Intruder | knows : H + (F,F') >
 => < X : Intruder | knows : H + (F,F') + F + F' > .

 rl [Composition1] :
 < X : Intruder | knows : H + A + M >
 => < X : Intruder | knows : H + (A <| M) + A + M > .

 rl [Composition2] :
 < X : Intruder | knows : H + F + F' >
 => < X : Intruder | knows : H + (F,F') + F + F' > .

 rl [AccessPublicKey] :
 < X : Intruder | knows : H + A >
 => < X : Intruder | knows : H + A + pk(A) > .
endom)
```