Rewriting Semantics of Meta-Objects and Composable Distributed Services [1]

G. Denker and J. Meseguer

Computer Science Laboratory
SRI International
Menlo Park, CA 94025
USA
Email: denker@csl.sri.com

C. Talcott

Computer Science Department
Stanford University
Stanford, CA 94305
USA
*Email:*clt@cs.stanford.edu

Abstract

Communication between distributed objects may have to be protected against random failures and malicious attacks; also, communication timeliness may be essential or highly desired. Therefore, a distributed application often has to be extended with communication services providing some kind of fault-tolerance, secrecy, or quality-of-service guarantees. Ideally, such services should be defined in a highly modular and dynamically composable way, so that the combined assurance of several services can be achieved by composition in certain cases, and so that services can be added or removed from applications at runtime in response to changes in the environment. To reason about the formal properties of such composable communication services one first needs to give them a precise semantics. This paper proposes a rewriting logic semantics for the so-called "onion skin" model of distributed object reflection, in which different *meta-objects*, providing different communication services, can be stacked on top of a basic application object. Since the correct behavior of a service depends on the type of *hostile environment* against which the service must protect the application, rewriting logic should also be used to specify such hostile environments. The service guarantees are then guarantees about the behavior specified by the union of the rewrite theories specifying the basic application, the services, and the hostile environment.

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

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 [15,10,17]), and also researchers in formal methods (c.f. [24,26,21]), have proposed some concepts and solutions to deal with the problems posed by composition of distributed services. The notion of "weaving" [14] 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). To achieve modularity and adaptability several of these approaches use distributed object-oriented reflection [2], that is, the ability to reprogram the semantics of communication and object creation. 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 [5] 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 [13]. However, the above approaches address what might be called parallel composition of systems. We are not aware of any systematic approach to the formal semantics of services composed in a reflective and layered way.

This paper proposes a semantic approach to make precise the reflective

[1] 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 N00014-99-C-0198, and by National Science Foundation Grants CCR-9505960 and CCR-9633363, and CCR-9900334.

[2] The specific features of distributed object-oriented reflection that we will discuss in this paper will not need the full power of rewriting logic reflection [7,8]; they can in fact be expressed at the object level without any need for using rewriting logic metalevel features.

concept of composable service in a distributed object-oriented system. Our approach is based on the executable formal semantics for distributed object-oriented systems provided by rewriting logic [19] and explicitly addresses the reflective distributed object-oriented 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 as a semantic foundation on top of which one could develop 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 [3,12,29,11,2,28,4] 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 a simple example (§1.2) to motivate the generalization and formalization of these ideas in the main body of the paper. In §2 we present a rewriting semantics of concurrent meta-objects. We begin with a review of the rewriting logic axiomatization of concurrent objects and specialize this to concurrent 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 concurrent objects can be transformed to equivalent single object towers that are then suitable for installation of additional meta-layers (§2.3). In §3 we present a case study illustrating our meta-object semantics. We use the rewriting logic based language Maude to specify our examples. The case study employs an encryption service and a client-server application in which the server may create helper objects and delegate requests to them. In §4 we explain how our work on composable services includes the formal specification of the hostile environment in which those services are installed. Conclusions and future work are discussed in §5.

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 its 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 environment 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.

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, statically or dynamically, 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 [11,28,25].

1.2 Example of Composing Services

In the onion skin architecture the implementation of a service is achieved by composing one or more meta-actors on top of each of the actors that comprise the basic distributed application. The meta-actors mediate the communication between the base actors by providing appropriate communication services. If several meta-actors are layered on top of a base actor, a combined communication service—for example providing both security and fault-tolerance—may thus be achieved. No knowledge of the component or modification of the base actor itself is needed. But, of course, for the meta-actors to implement the service correctly it may be necessary that the base actors satisfy some specific interface or behavioral requirements. To illustrate how these mechanisms can be used we sketch an example scenario.

Consider two actors A and B (Figure 1) where B provides some service requested by a client A.

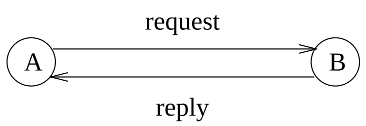

Fig. 1. A sends requests to B, and B replies and perhaps changes internal state.

Suppose, however, that sensitive information is being transmitted between A and B, and the communication medium cannot be trusted. Then, to protect the information, an *encryption* service can be put in place by installing encryptor meta-actors on A and B. These meta-actors take care of encrypting outgoing messages and decrypting incoming messages without any changes required in A and B (Figure 2).

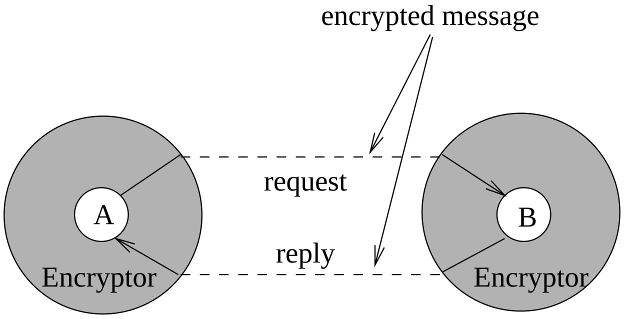

Fig. 2. All messages between A and B are encrypted and decrypted by the encryption layer.

This approach to composable services can also cope with the dynamic aspects of actor creation, including the installation of the appropriate meta-actors on such new actors. Suppose, for example, the variant of the above situation in which actor B may *delegate* the computation of the reply to a new actor B' that is created by B upon reception of the request from A with the only purpose of answering A's request (Figure 3).

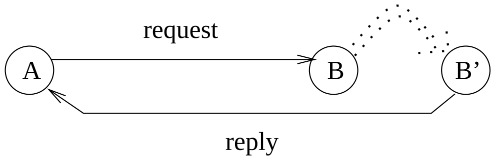

Fig. 3. B delegates request of A to B'. B' replies to A.

Of course, if only A and B used encryption but B' didn't, secrecy would be violated. A simple solution to this problem is to endow any encryption meta-actor with the capacity for installing another encryption meta-actor on any new actor that is being created by their "base" actor. In our example this means that the encryptor meta-actor for B will add to the object creation information that it receives from B for the creation of B' the additional information for adding an encryption meta-actor on top of B' (Figure 4).

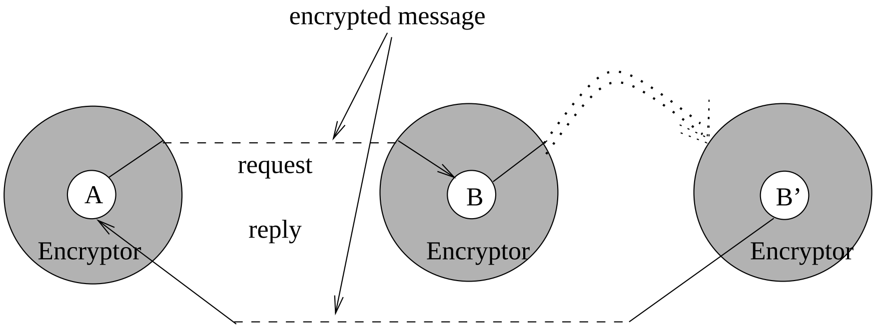

Fig. 4. B' is created with encryption service installed. B delegates request of A to helper B', and B' replies to A.

2 Rewriting Semantics of Concurrent Meta-Objects

We briefly introduce the rewriting logic model for a distributed system configuration as a "soup" (multiset) of concurrent objects and messages that behave according to a set of rewrite rules describing the behavior of individual objects. In particular, we specialize the rewriting logic object model [16,18,20] to the case of asynchronous message passing. We then introduce meta-objects, which can be layered to form towers that appear to their environment just like concurrent objects.

2.1 Concurrent Objects in Rewriting Logic

We adopt the treatment of concurrent objects by rewrite theories in [19]. We call such theories concurrent object theories. 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 $\_\ \_ : \mathbf{Configuration} \times \mathbf{Configuration} \longrightarrow \mathbf{Configuration}$. (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 ($\mathbf{Object}, \mathbf{Msg} < \mathbf{Configuration}$), and more complex configurations are generated from 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 describing the dynamics of an object-oriented system can have the form

$$\begin{aligned} r(\bar{x}) \ : \quad & M_1 \ldots M_n \ \langle O_1 : F_1 \mid atts_1 \rangle \ldots \langle O_m : F_m \mid atts_m \rangle \\ \longrightarrow \ & \langle O_{i_1} : F'_{i_1} \mid atts'_{i_1} \rangle \ldots \langle O_{i_k} : F'_{i_k} \mid atts'_{i_k} \rangle \\ & \langle Q_1 : D_1 \mid atts''_1 \rangle \ldots \langle Q_p : D_p \mid atts''_p \rangle \\ & M'_1 \ldots M'_q \quad \textit{if } \ \textit{Cond} \end{aligned}$$

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 *Cond* is the rule's condition[3]. 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 an *asynchronous object theory* if the rules are asynchronous, with exactly one object and at most one message on the lefthand side. In addition, we assume that message expressions 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*. An *asynchronous object configuration* is then a multiset of objects and message packets in the context of an asynchronous object rewrite theory.

2.2 Meta-Objects and Towers

Meta-objects are composable, 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. The communication between any two contiguous layers of a meta-object tower is synchronous. This is achieved by giving each meta-object four message lists: `out` and `in` lists for sending messages to and receiving messages from its upper level (another meta-object or the environment); and `up` and `down` lists for interaction with the lower level.

All meta-objects belong to a subclass of the class `MetaObject`, an object class with five distinguished attributes: `in`, `down`, `out`, `up`, `base`. The attribute `in` is a list of incoming messages representing messages to be delivered from the layer above or from the outside. The attribute `down` is a list of messages to be forwarded to the layer below. The attribute `out` is a list of configuration requests—that is, multisets of request packets—consisting of outgoing requests for message transmittal and object creation. The attribute `up` is a list of configuration requests that are to be forwarded to the layer above. 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` lists and place items in the `out` and `down` lists and modify other attributes. To interact with other meta-objects or with the en-

[3] We adopt the Maude convention that attributes that are not changed or used in rules do not need to be mentioned.

vironment a meta-object must be part of a meta-object tower. A meta-object tower is a structure of the form

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

built by means of an associative stack concatenation operation $\_\circ\_$ 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 of the tower 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.

$$\begin{array}{ll} \text{in:} & O_0 \lhd M\ \{\ \langle\ O_k : MC_k \mid \texttt{base} : O_0, \texttt{in} : L\ \rangle \circ MOS\ \} \\ & \quad \longrightarrow \{\ \langle\ O_k : MC_k \mid \texttt{base} : O_0, \texttt{in} : L \cdot M\ \rangle \circ MOS\ \} \end{array}$$

$$\begin{array}{ll} \text{out:} & \{\ \langle\ O_k : MC_k \mid \texttt{out} : \mathit{cfReq} \cdot cL\ \rangle \circ MOS\ \} \\ & \quad \longrightarrow \{\ \langle\ O_k : MC_k \mid \texttt{out} : cL\ \rangle \circ MOS\ \}\quad \mathit{CreateConf}(\mathit{cfReq}) \end{array}$$

$$\begin{array}{ll} \text{up:} & \langle\ O' : MC' \mid \texttt{up} : cL'\ \rangle \circ \langle\ O : MC \mid \texttt{out} : \mathit{cfReq} \cdot cL\ \rangle \\ & \quad \longrightarrow \langle\ O' : MC' \mid \texttt{up} : cL' \cdot \mathit{cfReq}\ \rangle \circ \langle\ O : MC \mid \texttt{out} : cL\ \rangle \end{array}$$

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

where MOS is a possibly empty lists of meta-objects, L, L' are lists of messages cL, cL' are lists of configuration requests, M is a message, cfReq is a message configuration containing requests for creation of a meta-object tower, and in the 'out' rule $\mathit{CreateConf}(\mathit{cfReq})$ is the meta-object tower configuration whose creation is requested by cfReq. For instance, the delegation application in §1.2 requests the creation of a clone of itself which will reply to a request of another object.

In order to deal with meta-object tower creation we have to extend the rules of meta-object towers. In §3.3 and §3.4 we propose rules for meta-objects and meta-object towers that can handle dynamic creation of meta-object tower configurations.

Figure 5 shows a meta-object tower with base object O, message packet $O \lhd M$ coming in from the surrounding environment, configuration $\mathit{CreateConf}(\mathit{cfReq})$ being created, message M' ready to move from the 'down' list of meta-object MO to the 'in' list below, and creation request cfReq' ready to move from the 'out' list of MO to the 'up' list above (possibly to the environment).

Note that, by remarking that the internal communication up and down a meta-object tower is invisible to outside observers, we can regard a meta-object tower with base object identifier O as behaving to the outside environment like an object with identifier O. Therefore, we can view the specification of any system in which meta-object towers communicate with each other by asynchronous message passing as a natural generalization of asynchronous object rewrite theories. We call the rewrite theories specifying such asynchronously communicating systems of meta-object towers *meta-object rewrite theories.*

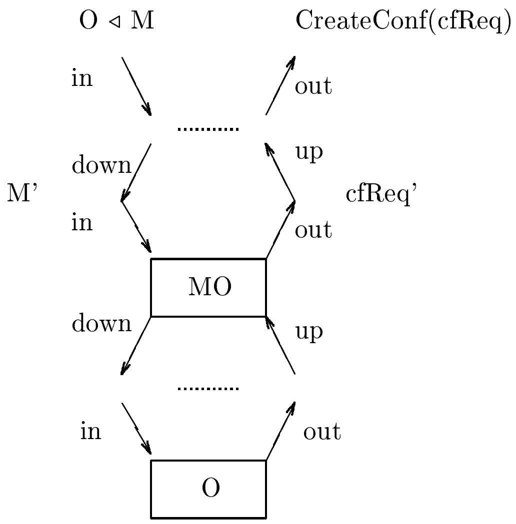

Fig. 5. A meta-object tower

2.3 Transforming Asynchronous Objects into Meta-Objects

We define a mapping from asynchronous object rewrite theories to meta-object rewrite theories, and show that this mapping preserves behavior. We assume an object theory in which the attribute identifiers of object classes are distinct from the distinguished meta-object attribute identifiers. To each asynchronous 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. An asynchronous object of the class is transformed into a single-layer meta-object tower as follows. Let $MOatts(O) = (\mathtt{base} : O, \mathtt{up} : \mathtt{nil}, \mathtt{dwn} : \mathtt{nil}, \mathtt{in} : \mathtt{nil}, \mathtt{out} : \mathtt{nil})$ 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 asynchronous object configurations (multisets 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

$$r\text{:} \quad O \lhd M \ \ \langle O : OC \mid atts \rangle \longrightarrow \langle O : OC \mid atts' \rangle \ \ oCf$$

where oCf is an asynchronous object configuration, is mapped to the meta-object rule

$$\begin{aligned} \mathit{o2m}(r)\text{:}\quad & \langle O : \mathit{o2m}(OC) \mid \mathit{atts}, \mathtt{out} : cL, \mathtt{in} : M \cdot L \rangle \\ & \longrightarrow \langle O : \mathit{o2m}(OC) \mid \mathit{atts}', \mathtt{out} : cL \cdot \mathit{o2r}(\mathit{oCf}), \mathtt{in} : L \rangle \end{aligned}$$

where $\mathit{o2r}(\mathit{oCf})$ is the request for creation of $\mathit{o2m}(\mathit{oCf})$:

$$\mathit{CreateConf}(\mathit{o2r}(\mathit{oCf})) = \mathit{o2m}(\mathit{oCf}).$$

Theorem 2.1 *Given any asynchronous object class OC, in the union of the rewrite theories for the classes OC and $\mathit{o2m}(OC)$ the objects*

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

are observationally equivalent. More generally, let oCf be any asynchronous object configuration then oCf and $\mathit{o2m}(\mathit{oCf})$ are observationally equivalent configurations.

To make sense of the above theorem we need to say what is meant by observational equivalence. We say that two asynchronous object configurations are observationally equivalent if they have the same set of observations. There are many possible choices of observation. Here we pick a simple one that records the sequence of messages delivered. Let γ be a computation (proof term) (starting from an object configuration oCf) that is in sequential form, that is γ is a sequence of single rewrite rule applications $r(\theta)$. In the case of simple asynchronous object theories, we say that application of $r(\theta)$ delivers the message $O \triangleleft M$ if the rule instance has the form:

$$r(\theta) : \quad O \triangleleft M \;\; \langle O : OC \mid \mathit{atts} \rangle \longrightarrow \langle O : OC \mid \mathit{atts}' \rangle \;\; \mathit{oCf}$$

In the case of meta-object theories, we say that the application of $r(\theta)$ delivers the message $O \triangleleft M$ if the rule instance has the form:

$$\begin{aligned} r(\theta) : \quad & \langle O : MC \mid \mathit{atts}, \mathtt{out} : cL, \mathtt{in} : M \cdot L \rangle \\ & \longrightarrow \langle O : MC \mid \mathit{atts}', \mathtt{out} : cL \cdot \mathit{o2r}(\mathit{oCf}), \mathtt{in} : L \rangle \end{aligned}$$

For a rewrite rule application $r(\theta)$, we define $\mathit{obs}(r(\theta))$ to be the singleton sequence containing the message $O \triangleleft M$ delivered, if any, and the empty sequence otherwise. The observation $\mathit{obs}(\gamma)$ of a rewrite computation γ is then the concatenation of the observations of the basic rewrites of γ, and $\mathit{obs}(\mathit{oCf})$ is the set of observations for all finite computations with source oCf. Two asynchronous object configurations are then observationally equivalent if they have the same set of observations. This notion of observation is similar to one studied in [30], and is also related to the partial order of event semantics for asynchronous object theories of [20].

Proof sketch: The idea is to extend $\mathit{o2m}$ inductively to sequential computations with source oCf by mapping $r(\theta)$ to $\alpha_{in}; \mathit{o2m}(r)(\theta'); \alpha_{out}$, where the input/output rule instances α_{in}, α_{out} can easily be computed from $r(\theta)$, and where θ' extends the substitution θ to take care of matching the meta-object class attributes appropriately. It is easy to see that $obs(\gamma) = obs(o2m(\gamma))$. Thus $obs(oCf) \subseteq obs(o2m(oCf))$. For the other direction we note that if γ' is any sequential computation of $o2m(oCf)$, then there is a computation γ_c such that $obs(\gamma') = obs(\gamma_c)$ and γ_c has a canonical form in which the internal queues effectively remain empty and each rule application $o2m(r)(\theta)$ appears in a context $\alpha_{in}; o2m(r)(\theta); \alpha_{out}$, where α_{in} is the delivery of the message consumed by $r(\theta)$ if any, and where α_{out} emits the configuration elements created by $r(\theta)$. From γ_c one can obtain a computation γ of oCf such that $obs(\gamma) = obs(o2m(\gamma))$.

3 A Composable Communication Service Example

In the following we illustrate how the encryption service and the delegating client-server application of §1.2 are formalized in our semantics.

Due to space limitations we cannot present the complete Maude specification. However, we provide the main parts of the specification. The following modules make use of some of Maude's builtin modules and theories, such as the sort parameter theory `TRIV`, and the modules, `BOOL` for booleans with sort `Bool`, and `MACHINE-INT` for integers, with sort `MachineInt`, and the Full Maude module `CONFIGURATION` with sort `Configuration` for multisets of objects and messages. We also assume a module `METAOBJECT` with class `MetaObject` and sorts `MetaStack` and `MetaTower` (subsorts of `Configuration`) that represents meta-objects and meta-object towers following closely the presentation in §2.2.

3.1 Security Service

As an example (meta)service, a simple security service is specified. The meta-object class `EncrService` below specifies a composable service designed to provide secure message transmission in the presence of intruders that may observe, intercept, or fake messages. The encryption service uses *public-key encryption*. The data being encrypted is called a "field" (`Field`) and `Pkey` is the sort for keys. `op {_}_ : Field PKey -> Field` is the de/encryption operator. A key K1 that that constitutes a keypair with another key K2 can be used to decrypt a field that has been encrypted using K2. That is the following axiom holds `ceq {{F}PK}SK = F if keypair(PK,SK)` . We assume a module `CRYPTO-DT` in which all cryptographic operators, sorts and axioms are specified.

```
( 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[ConfReq] .
  var M : Msg .

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

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

3.2 A Simple Client-Server Application

A simple client-server application is specified as an ordinary asynchronous object system module `SIMPLEAPP`. In the module `SIMPLEAPP`, the class `SimpleClient` models a client that has a list of data items for which it will request its server to carry-out a simple operation and wait for a reply. The class `SimpleServer` replies to such requests by applying a function `f` and replying with the resulting data.

```
(fth DATA is sort Data . endfth)

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

( omod SIMPLEAPP is
   inc DATA .
   protecting LIST  * (op __ to _*_) [Data] .

   class SimpleClient | tasks : List[Data] , ready : Bool , server : Oid .

   op reply : Data -> Msg .
   op simple : Data Oid -> Msg .

   vars X Y : Oid .  var D : Data .  var dL : List[Data] .

   rl [sendSimple] :
   < X : SimpleClient | tasks : ( D * dL ) , ready : true , server : Y >
     =>
   < X : SimpleClient | tasks :  dL , ready : false > ( Y <| simple(D,X) ) .

   rl [rcvReply] :
   < X : SimpleClient | ready : false > ( X <| reply(D) )
     =>
   < X : SimpleClient | ready : true > .

   class SimpleServer | .
   op f : Data -> Data .

   rl [rcvSimple] :
   < Y : SimpleServer |  > ( Y <| simple(D,X) )
     =>
   < Y : SimpleServer | >  ( X <| reply(f(D)) ) .

   *** Creating an initial Client-Server configuration iCf
   op oA oB  : Oid .
   op d1 d2 d3 : Data .
   op server0  : Oid -> Object .
   op client0 : Oid List[Data] Oid -> Object .
   op iCf : Configuration .
```

```
   eq server0(Y) = < Y : SimpleServer | > .
   eq client0(X,dL,Y) =
      < X : SimpleClient | tasks : dL , ready : true , server : Y > .

   eq iCf = client0(oA,(d1 * d2 * d3),oB) server0(oB) .
endom )
```

To make the client and server composable, we apply the *o2m* transformation of §2.3 to `SIMPLEAPP`, obtaining `SIMPLEAPPMO`. The module `SAMPLECONF` composes the encryption service with the Client-Server application. Now communication between the two is secure.

```
( omod SIMPLEAPPMO is
   inc DATA .
   inc METAOBJECT .
   protecting LIST  * (op __ to _*_) [Data] .

   class SimpleClientMO | tasks : List[Data] , ready : Bool , server : Oid .
   subclass SimpleClientMO < MetaObject .

   op reply : Data -> Msg .
   op simple : Data Oid -> Msg .

   vars X Y : Oid .   var D : Data .  var dL : List[Data] .
   var cL : List[ConfReq] .  var L : List[Msg] .

   rl [sendSimple] :
   < X : SimpleClientMO | tasks : (D * dL), ready : true, server : Y, up : cL >
    =>
   < X : SimpleClientMO | tasks :  dL , ready : false,
                      up : cL * ( Y <| simple(D,X) ) > .

   rl [rcvReply] :
   < X : SimpleClientMO | ready : false , in : ( X <| reply(D) ) ; L
     =>
   < X : SimpleClientMO | ready : true , in : L > .

   class SimpleServerMO | .
   op f : Data -> Data .

   rl [rcvSimple] :
   < Y : SimpleServerMO |  in :  ( Y <| simple(D,X) ) * L , out : cL  >
     =>
   < Y : SimpleServerMO | in : L , out : cL * ( X <| reply(f(D)) ) >.

   op oA oB  : Oid .
   op d1 d2 d3 : Data .
   op serverMO  : Oid -> MOStack .
   op clientMO : Oid List[Data] Oid -> MOStack .

   eq serverMO(Y) = < Y : SimpleServerMO | moAtts(Y) > .
   eq clientMO(X,dL,Y) =
   < X : SimpleClientMO | tasks : dL , ready : true , server : Y,  moAtts(X) > .
endom )

( omod SAMPLECONF is
  inc ENCR .
  inc SIMPLEAPPMO .

  op motA motB : MOTower .
  op moConf : Configuration .
  op moA moB : Oid .

  eq  motA =
    { < moA : EncrService | moAtts(oA) > o clientMO(oA, (d1 * d2 * d3), oB) } .
  eq  motB = { < moB : EncrService | moAtts(oB) > o serverMO(oB) } .
```

```
  eq moConf = motA motB .
  endom )
```

The configuration `moConf = motA motB` corresponds to the picture in Figure 2.

3.3 Adding dynamic object creation to Client-Server application

Now the functionality of the SimpleServer is extended to handle complex requests by creating an object of the `Helper` class to do the computation, thus freeing the server to answer other requests. The `SimpleClient` class is also extended to make complex requests.

To treat dynamic objects (objects that may create other objects) in a uniform way, we introduce the class `ObjectD` and an operation `newId` for locally generating fresh object identifiers for newly created objects. Objects of class`ObjectD` have an attribute, `newCnt : MachineInt`, used to track the number of objects created by this object.

```
( omod OBJECTD is
  class ObjectD | newCnt : MachineInt .
  op newId : Oid MachineInt -> Oid .
endom )
```

The client class is extended with the ability to send complex requests, and the server class is extended by adding a rule for serving complex requests. It is subclassed with `ObjectD` in order to create helper objects to carry out the complex computation.

```
( omod DELEGATINGAPP is
  inc SIMPLEAPP .
  inc OBJECTD .

  class ComplexClient | ctasks : List[Data] .
  subclass ComplexClient < SimpleClient .

  op complex : Data Oid -> Msg .

  vars X Y Z : Oid .  var D : Data .  var dL : List[Data] .

  rl [sendComplex] :
  < X : ComplexClient | ctasks : D . dL , ready : true , server : Y >
    =>
  < X : ComplexClient | ctasks :  dL , ready : false > ( Y <| complex(D,X) ).

  class ComplexServer | .
  subclass ComplexServer < SimpleServer .
  subclass ComplexServer < ObjectD .


  rl [rcvComplex] :
  < Y : ComplexServer | newCnt : n > ( Y <| complex(D,X) )
    =>
  < Y : ComplexServer | newCnt : n+1 >
    < newId(Y,n) : Helper | task : D , customer : X , done : false >   .

  class Helper | task : Data, customer : Oid, done : Bool .
  op g : Data -> Data .

  rl [doComplex] :
```

```
    < Z : Helper | task : D , customer : X , done : false >
    =>
    < Z : Helper | task : D , customer : X , done : true > ( X <| g(D) ) .
endom )
```

This object module is transformed into a meta-object module as before.

3.4 Adding dynamic object creation to MetaObjects

We extend the `METAOBJECT` module with syntax for creation requests and define the class `MetaObjectD` of dynamic meta-objects by inheritance from the classes `MetaObject` and `ObjectD`.

```
( omod DYNMETAOBJECT is
  inc METAOBJECT .
  inc OBJECTD .

  class MetaObjectD |  .
  subclass MetaObjectD < MetaObject .
  subclass MetaObjectD < ObjectD .

*** Extending the tower model with syntax to deal with tower creation

  sort TowerReq .
  op BaseO : Oid Cid AttributeSet -> TowerReq .
  op AddMeta : Oid Cid AttributeSet TowerReq -> TowerReq .

  subsort  TowerReq < ConfReq .

  op createStack : TowerReq -> Tower .
  op baseId : TowerReq -> Oid .

  vars  cfR cfR0 cfR1 : ConfReq .
  var  M : Msg .
  vars  X Y : Oid .
  vars  MC : MetaObject .
  var T : TowerReq .
  var ATT : AttributeSet .

  eq createConf(M) = M .
  eq createConf(cfR0 * cfR1) = createConf(cfR0) createConf(cfR1) .
  eq createConf(T) = { createStack(T) } .

  eq createStack(BaseO(X, MC, ATT)) = < X : MC | ATT, newCnt : 0, moAtts(X) > .
  eq createStack(AddMeta(X, MC, ATT, T ))
    =  < X : MC | ATT, newCont : 0,  moAtts(baseId(T)) > o createStack(T)

  eq  baseId(BaseO(X, MC, ATT)) = X .
  eq  baseId(AddMeta(X, MC, ATT, T)) = baseId(T) .
endom )
```

3.5 Composing Encryption with the Delegating Client Server

We first apply the *o2m* transformation to the module `DELEGATINGAPP` to obtain the module `DELEGATINGAPPMO` with the corresponding composable client and server classes. The module `ENCRD` extends the encryption service to install itself on created towers before sending the request up the stack. The extension is accomplished by subclassing with `MetaObjectD` and adding an additional rule for tower requests. For simplicity we assume that what comes in the `up` list is either a single message or a single tower creation request.

```
( omod DELEGATINGAPPMO is
  inc SIMPLEAPPMO .
  inc DYNMETAOBJECT .

  class ComplexClientMO | ctasks : List[Data] .
  subclass ComplexClientMO < SimpleClientMO .
  subclass ComplexClientMO < MetaObjectD .

  op complex : Data Oid -> Msg .

  vars X Y Z : Oid .  var D : Data .
  var dL : List[Data] .   var cL : List[ConfReq] .  var L : List[Msg] .

  rl [sendComplex] :
  < X : ComplexClientMO | ctasks : D . dL, ready : true, server : Y, out : cL >
  =>
  < X : ComplexClientMO | ctasks :  dL, ready : false, base : X,
                          out : cL * ( Y <| complex(D,X) ) > .

  class ComplexServerMO |  .
  subclass ComplexServerMO < SimpleServer .
  subclass ComplexServerMO < ObjectD .
  subclass ComplexServerMO < MetaObjectD .

  rl [rcvComplex] :
  < Y : ComplexServerMO | newCnt : n,
                          in : L ; ( Y <| complex(D,X) ) , out : cL >
    =>
  < Y : ComplexServerMO | newCnt : n+1 , in : L ,
    out : cL * BaseO(newId(Y,n), HelperMO, (task : D, customer : X, done : false)) >  .

  class HelperMO | .
  subclass HelperMO < Helper .
  subclass HelperMO < MetaObject .
  op g : Data -> Data .

  rl [doComplex] :
  < Z : HelperMO | task : D , customer : X , done : false , out : cL >
  =>
  < Z : HelperMO | task : D , customer : X , done : true
                        out : cL * ( X <| g(D) ) > .
endom )

( omod ENCRD is
  protecting ENCR .
  inc DYNMETAOBJECT .

  class EncrDService .
  subclass EncrDService < MetaObjectD .
  subclass EncrDService < EncrService .

  vars X A B : Oid .  vars cL cL' : List[ConfReq] .
  vars PK SK : PKey .  var S : PSeal .
  var M : Msg .  var N : MachineInt .  var T : TowerReq .

  rl [createWithEncr] :
    < MO : EncrDService | newCnt : N,  up : T . cL , out : cL'>
  ==>
  < MO : EncrDService | newCnt : N+1,  up : cL,
        out : cL' . AddMeta(newId(X,N),EncrDService,newCnt : 0,T)  > .
 endom )
```

4 Hostile Environments

The reason why we need communication services in the first place is because the world in not perfect. The communication medium, typically realized by some network, in general is neither secure nor fault-free.

In general, all this means that the communication medium may contain *hostile elements*—either unintentional, e.g., faults, or malicious, e.g., an intruder—and will have *resource and performance limitations*. Specific communications services make sense as solutions to specific assumptions about given hostile environments and/or performance limitations, and typically make some *guarantees* about the correct behavior of the given application, in the sense that, roughly, the service allows it to function *as if* the given problem did not exist.

The point is that we can formally specify in rewriting logic both the services and the assumptions about the communication environment.

In our running example, we could for example specify a class of *intruder* objects that can, say, intercept and replay messages by giving appropriate rewrite rules for such actions. We could model the knowledge of an intruder as one particular attribute, in which every intercepted message is stored. An intruder can replay stored messages at any time.

```
( omod INTRUDER is
  protecting SET * ( op __ to _+_) [Item] .

  class Intruder | knows : Set[Item] .

  vars A X : Oid . var M : Msg . var H : Set[Item] .

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

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

Even if such an intruder were not able to decrypt messages between A, B, and B’, it could nevertheless cause other trouble, such as stealing some messages, and fooling A by replaying replies from B to previous requests, so that the guarantees we can make are still limited. This would then suggest adding a fault-tolerant service layer to ensure in-order/deliver-once fault-tolerant communication.

5 Conclusions and Future Work

We have given a rewriting logic semantics to an important form of distributed object-oriented reflection that permits composing different distributed services in a modular and principled way. We have illustrated the ideas by means of an encryption service and a client-server application in which the server may create helpers and delegate selected tasks to them.

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. In this first step we have focused on developing adequate semantic foundations. Based on these foundations we plan to investigate in the future property-oriented formalisms adequate to express and prove *service guarantees* about composable communication services, relative to given assumptions about the hostile environments in which the services are placed and about the other services or applications with which they are composed.

Although our formalism provides a means to install new service layers, there is still a need to reason about the possible interactions between the layers. For example, consider two meta-objects providing two different services such as security and fault-tolerance that can be stacked on top of a basic application. The interesting question is which of the two different composition orders achieves the desired combination of properties. For example, stacking the fault-tolerance service on top of the encryption service will 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. If we place the fault-tolerant service as the top layer, the information used in the fault-tolerant service is not protected, and thus, could be changed by a malicious environment: for example, an acknowledgement could be faked. Then the guarantees of the fault-tolerance service will no longer hold true. However, under reasonable assumptions, a secure and reliable communication service can be achieved by the alternate composition that places the security service as the top layer.

To specify and reason about service guarantees such as the above, we plan to investigate a variety of formalisms—including different temporal and modal logics. Even a simple first-order extension of rewriting logic with Boolean connectives and quantifiers is sufficient to state a number of useful properties. 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 and stochastic properties may be quite important, and one would then like to use a formalism supporting real-time aspects.

Much more work remains ahead, particularly in the following aspects:

- 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, taking into account work such as [23,9,6].
- Different meta-object composition mechanisms should be investigated, including a more detailed study of the relations between our work and that of Agha, Astley, and Sturman [28,4]. Also, 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. Furthermore, besides point-to-point communication we should investigate broadcast and multicast services.

- 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. [27]).
- Performance issues should also be dealt with by specifying in sufficient detail the appropriate resources, such as buffers with appropriate sizes, channels, cpu cycles, network bandwidth, and so on; and by making use of the possibilities for rewriting logic for modeling and analyzing real-time system aspects [22].

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

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

[7] M. Clavel. Reflection in general logics and in rewriting logic, with applications to the Maude language. Ph.D. Thesis, University of Navarre, 1998.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer. Metalevel computation in Maude. *Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1998.

[9] 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, 1997. LNCS 1275.

[10] D. Feldmeier, A. McAuley, J. Smith, D. Bakin, W. Marcus, and T. Raleigh. Protocol Boosters, 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.

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

[12] 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.

[13] 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.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. 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, URL: `http://www.parc.xerox.com/spl/groups/eca/pubs/papers/Kiczales-ECOOP97/`.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] J. Meseguer and C. Talcott. A Partial Order Event Model for Concurrent Objects. In J. Baeten and S. Mauw, editors, *Proc. 10th Intern. Conf. on Concurrency Theory (CONCUR'99), Eindhoven, The Netherlands, August 1999*, pages 415–430. Springer, 1999. LNCS 1664.

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

[22] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. Submitted for publication. `http://maude.csl.sri.com`, 1999.

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

[24] 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.

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

[26] 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`.

[27] S. F. Smith and C. L. Talcott. Modular reasoning for actor specification diagrams. In P. Ciancariani, A. Fantechi, and R. Gorrieri, editors, *Formal Methods for Open Object-based Distributed Systems*, pages 313–330. Kluwer, 1999.

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

[29] 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.

[30] N. Venkatasubramanian and C. L. Talcott. Integration of resource management activities in distributed systems. Technical report, 2000. in preparation.