Semantic Models for Distributed Object Reflection

José Meseguer[1] and Carolyn Talcott[2]

[1] Computer Science Department, University of Illinois at Urbana-Champaign, Urbana IL 61801

[2] Computer Science Laboratory, SRI International, Menlo Park, CA 94025

Abstract. A generic formal model of distributed object reflection is proposed, that combines logical reflection with a structuring of distributed objects as nested configurations of metaobject that can control subobjects under them. The model provides mathematical models for a good number of existing models of distributed reflection and of reflective middleware. To illustrate the ideas, we show in some detail how two important models of distributed actor reflection can be naturally obtained as special cases of our generic model, and discuss how several recent models of reflective middleware can be likewise formalized as instances of our model.

1 Introduction

Distributed object reflection is a crucial technique for the development of a next generation of adaptive middleware systems, mobile languages, active networks, and ubiquitous embedded distributed systems. Semantic models for distributed object reflection can serve two important purposes. First of all, they can contribute to the *conceptual sharpening and unification* of reflective notions, so that existing reflective systems become easier to understand and, more importantly, so that the design of new systems can be done in a clear, simple, and principled way. A second important concern is *high assurance*, which is harder to achieve because of the complexity added by reflection to already complex distributed systems. This concern is real enough, because reflective systems, for all their nice properties, can become *Trojan horses* through which system security could be compromised in potentially devastating ways. Without semantic models any *mathematical* verification of highly critical system properties is of course impossible.

In this paper we propose a generic formal model of distributed object reflection that seems very flexible: it yields as special cases mathematical models for a good number of existing models of distributed reflection and of reflective middleware. The model is based on a simple *executable logic* for distributed system specification, namely rewriting logic [51, 52]. Rewriting logic has high performance implementations [19, 15], as well as an environment of formal tools including theorem provers [21], and model checkers such as the Maude LTL model checker. Therefore, the semantic model of a reflective distributed object system developed using the methods proposed here can be symbolically simulated, and can be formally analyzed using model checking and theorem proving tools.

One important feature of our generic model is the powerful combination of *distributed object reflection* and *logical reflection* that it provides. We discuss logical reflection in Section 3. In the rest of this introduction we first discuss earlier work on object-oriented reflection (Section 1.1) and on semantic models of distributed object reflection (Section 1.2). Then we explain in more detail the contributions of this paper (Section 1.3).

1.1 Object-Oriented Reflection

Research on computational reflection was initiated by work of Brian Smith [63] (3-Lisp) and Patty Maes [47] (3KRS). This work introduced ideas such as towers of reflection, and causal connection. An overview of research in computational reflection can be found in proceedings of several recent workshops and conferences [78, 40, 24, 77, 44].

Reflective programming languages. A number of reflective actor-based languages have been developed to support separation of concerns and high-level programming abstractions for distributed systems. The ABCL family of languages [76] explores different forms of reflection, including single-actor and group-based reflection. A layered reflection model (the *onion skin model*) [4, 3] models different concerns such as application functionality, communication protocols and security requirements, and failure/dependability semantics 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 [32], actor spaces [2], real-time synchronizers [58], protocols that abstract over interaction patterns [66], and dynamic architectures [9]. Metaobject protocols [41] provide more restricted forms of reflective capability, providing interfaces to a language that give users the ability to incrementally modify the language's behavior and implementation. This approach has been generalized to the Aspect Oriented Programming paradigm [42] to facilitate separation of concerns, composition, and re-use in programming.

Operating systems. A number of operating systems build on reflective distributed object models, thus allowing application objects to customize the system behavior. In the AL-D/1 reflective programming system [56] an object is represented by multiple models, allowing behavior to be described at different levels of abstraction and from different points of view. Additional examples are Apertos [38], Legion [34], and 2K [45].

Reflective middleware. Adaptability and extensibility are prime requirements of middleware systems, and several groups are doing research on reflective middleware [44]. Reflective middleware typically builds on the idea of a metaobject protocol, with a metalevel describing the internal architecture of the middleware, and reflection used to inspect and modify internal components. DynamicTao [43] is a reflective CORBA ORB [55] built as an extension of the Tao real-time CORBA ORB [61]. DynamicTao supports on-the-fly reconfiguration while maintaining consistency by reifying both internal structure and dependency relations using objects called configurators. In [74] the use of reflective middleware techniques to enhance adaptability in Quality of Service (QoS)-enabled component-based applications is discussed and illustrated using the Tao ORB. The distributed Multimedia Research Group at Lancaster University has proposed a reflective architecture for next-generation middleware based on multiple metamodels [13, 12], and a prototype has been developed using the reflective capabilities of Python.

Middleware systems often contain components that are reflectively related to the application level and/or the underlying infrastructure. For example Quo [79, 46] has *system condition objects* that provide interfaces to resources, mechanisms, orbs etc. that need to be observed, measured or controlled. Delegates reify method requests and evaluate them according to *contracts* that represent strategies for meeting service level agreements. Another example is the Grid Protocol architecture proposed in [31], in which the resource level contains protocols for query and control of individual resources.

1.2 Semantic Models of Distributed Object Reflection

Most of the work on computational reflection has focused on development of reflection mechanisms, and on design and use of reflective systems to achieve a variety of goals such as separation of concerns, extensibility, and dynamic adaptability. Much less work has been done on the underlying theory, although some initial efforts have been made toward developing semantic models, principles for reasoning and techniques for analysis.

A formal model of the ODP object reference model based on rewriting logic is given in [54], addressing key issues such as object binding. A translation of formal QoS specifications into monitors and controllers is discussed in [14]. In [48] a metaobject protocol for CORBA objects is formalized using the π-calculus. Such a formal model can be used to generate execution traces, and can be analyzed using existing tools for properties such as deadlock freedom. In [8] CSP is used to give a formal semantics to aspects and aspect weaving.

Two reflective architectures for actor computation have been used as a basis for defining and reasoning about composable services in dynamic adaptable distributed systems: the *onion skin* model and the *two-level actor machine (TLAM)* model. A formal executable semantics for the onion skin model of reflection is given in [26]. The TLAM [72, 68, 73, 70] is a semantic framework for specifying, composing and reasoning about resource management services in open distributed systems. The two levels provide a clean separation of concerns and a natural basis for modeling and reasoning about customizable middleware and its integration with application activity. This model has been used to reason about distributed garbage collection [69], the safe composition of system-level activities such as remote creation, migration, and recording of global snapshots [71, 72], and QoS-based multimedia services [68, 70].

Maude has been used to formalize aspects of several existing distributed system standards. A method for integration with CORBA components is explained in [7] and Maude interaction with existing system components using SOAP is described in [6]. Using these ideas Maude executable specifications can plug and play with system components implemented on arbitrary platforms. Formal modeling of ODP enterprise and information viewpoints is illustrated in [28, 29].

1.3 This Paper

Two important ingredients of our approach are the rewriting logic representation of distributed object systems (explained in Section 2) and the concept of logical reflection, including a logic-independent axiomatization of reflective logics in general, and rewriting logic reflection in particular (Section 3). Our generic model is then explained in Section 4. It combines two ideas:

1. a "Russian dolls" idea, in which distributed objects are structured in nested configurations of metaobjects that can control subojects under them, with an arbitrary number of levels of nesting, so that a given metaobject may itself be a subobject of a metametaobject, and so on;
2. the use of logical reflection, so that subobjects may at times be metarepresented as *data*, again with an arbitrary number of levels of nesting; this, combined with the Russian dolls idea, greatly increases the reflective capabilities of a system, allowing, for example, a simple design of reflective systems that can be both mobile and adaptive.

The usefulness of any model has to be shown in its applications. We therefore show in detail how two important models of distributed actor reflection, the onion skin model, and the two-level actor machine model (TLAM) can be naturally obtained as special cases of our generic model (Section 5). Similarly, we discuss in Section 6 how several recent models of reflective middleware can be formalized in terms of our generic model. We finish the paper with some concluding remarks in Section 7.

2 Modeling Distributed Objects in Rewriting Logic

In this section we explain how distributed object systems are axiomatized in rewriting logic. In general, a rewrite theory is a triple $\mathcal{R} = (\Sigma, E, R)$, with (Σ, E) an equational specification with signature of operators Σ and a set of equational axioms E; and with R a collection of labelled rewrite rules. The equational specification describes the *static* structure of the distributed 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 E.

Let us see in more detail how the state space of a distributed object system can be axiomatized as the initial algebra of an equational theory (Σ, E). That is, we need to explain the key state-building operators in Σ and the equations E 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. As we shall see in Section 4, there can be more general ways of structuring the distributed state than just as a *flat* multiset of objects and messages; however, the flat multiset structure is the simplest and will help us explain the basic ideas. Assuming such a structure, we can view configurations as built up by a binary multiset union operator which we can represent with empty syntax (i.e., juxtaposition) as

$$\_\ \_ : \texttt{Configuration} \times \texttt{Configuration} \longrightarrow \texttt{Configuration}.$$

(Following the conventions of mix-fix notation, underscore symbols (_) are used to indicate argument positions.) The 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 `Object` `Msg` $<$ `Configuration`, 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 names of 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 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. This finishes the description of some of the sorts, operators, and equations in the theory (Σ, E) axiomatizing the states of a concurrent object system. Particular systems will have additional operators and equations, specifying, for example, the data operators on attribute values, and perhaps other state-building operators besides multiset union.

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

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

where r is the label, 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*. The above rule format assumes again a *flat* multiset configuration of objects and messages. We shall see in Sections 4-5 more general rules specifying object interactions in nonflat state configurations.

For example, we can consider three classes of objects, `Buffer`, `Sender`, and `Receiver`. The buffer stores a list of numbers in its `q` attribute. Lists of numbers are built using an associative list concatenation operator, `_._` with identity `nil`, and numbers are regarded as lists of length one. The name of the object reading from the buffer is stored in its `reader` attribute. The sender and receiver objects store a number in a `cell` attribute that can also be empty (`mt`) and have also a counter (`cnt`) attribute. The sender stores also the name of the receiver in an additional attribute. The counter attribute is used to ensure that messages are received by the receiver in the same order as they are sent by the sender even though communication between the two parties is asynchronous. Each time the sender gets a new message from the buffer, it increments its counter. It uses the current value of the counter to tag the message sent to the receiver. The receiver only accepts a message whose tag is its current counter. It then increments its counter indicating that it is ready for the next message. Using Maude syntax [19, 20], the three classes above are defined by declaring the name of the class, followed by a "`|`", followed by a list of pairs giving the names of attributes and corresponding value sorts. Thus we have

```
class Buffer | q: List[Nat], reader: OId .
class Sender | cell: Default[Nat], cnt: Nat, receiver: OId .
class Receiver | cell: Default[Nat], cnt: Nat .
```

where `OId` is the sort of *object identifiers*, `List[Nat]` is the sort of lists of natural numbers, and `Default[Nat]` is a supersort of `Nat` adding the constant `mt`. Then, three typical rewrite rules for objects in these classes (where `E` and `N` range over natural numbers, `L` over lists of numbers, `L.E` is a list with last element `E`, and `(to Z : E from (Y,N))` is a message) are

```
rl [read] : < X : Buffer | q: L . E, reader: Y >
                 < Y : Sender | cell: mt, cnt: N >
              => < X : Buffer | q: L, reader: Y >
                 < Y : Sender | cell: E, cnt: N + 1 > .

rl [send] : < Y : Sender | cell: E, cnt: N, receiver: Z >
    => < Y : Sender | cell: mt, cnt: N > (to Z : E from (Y,N)) .

rl [receive] : < Z : Receiver | cell: mt, cnt: N >
                  (to Z : E from (Y,N))
                 => < Z : Receiver | cell: E, cnt: N + 1 > .
```

where the `read` rule is synchronous and the `send` and `receive` rules asynchronous. These rules are applied *modulo* the associativity and commutativity of the multiset union operator, and therefore allow both object synchronization and message sending and receiving events anywhere in the configuration, regardless of the position of the objects and messages. We can then consider the rewrite theory $\mathcal{R} = (\Sigma, E, R)$ axiomatizing the object system with these three object classes, and with R the three rules above (and perhaps other rules, such as one for the receiver to write its contents into another buffer object, that we omit).

Rewriting logic [51] then gives a simple inference system to deduce, for a system axiomatized by a rewrite theory $\mathcal{R}$, all the finitary concurrent computations possible in such a system. Such computations are identified with *proofs* of the general form $\alpha : t \longrightarrow t'$ in the logic. The intuitive idea is that such proofs/computations correspond to finitary concurrent behaviors of the distributed system so axiomatized. They are described as *concurrent rewritings*, where several rules may fire simultaneously in the distributed state, can be followed by other such simultaneous firing of other rules, and so on.

For example, a buffer object `a`, and sender and receiver objects `b` and `c` can be involved in a concurrent computation in which `b` reads a value from `a` and sends it to `c`, and then, simultaneously, `c` receives it and `b` reads a second value from `a`. Suppose that we begin with the following initial configuration C_0

```
< a : Buffer | q: 7 . 9, reader: b >
< c : Receiver | cell: mt, cnt: 1 >
< b : Sender | cell: mt, cnt: 0, receiver : c >
```

Then, the configuration C_0 is transformed by the above-mentioned concurrent rewriting into the following final configuration C_1:

```
< a : Buffer | q: nil, reader: b >
< b : Sender | cell: 7, cnt: 2, receiver : c >
< c : Receiver | cell: 9, cnt: 2 >
```

Under reasonable assumptions about the rewrite theory $\mathcal{R}$, such concurrent rewritings can be *executed*, either by *simulating* the concurrent rewriting as multiset rewriting in a sequential implementation of rewriting logic such as Maude [19, 20], ELAN [15], or CafeOBJ [33]; or by *distributed execution* in a language such as Mobile Maude [27] in which the rewrite rules *are* the distributed code.

3 Logical Reflection

In logic, reflection has been studied by many researchers since the fundamental work of Gödel and Tarski (see the surveys [64, 65]). One strand of computer science research on reflection is related, either implicitly or explicitly, to the logical understanding of reflection. The two areas where this strand has been mostly developed are: (1) declarative programming languages, where logical reflection is used in the form of *metacircular interpreters* [59]; and (2) theorem proving, where logical reflection can be used to increase in a disciplined and sound way the deductive power of theorem provers.

Different logics may be involved. For declarative languages one could mention, among others, reflective language designs based on the pure lambda calculus [53], equational logic [67], and Horn logic [36]. In theorem proving there has been a substantial body of research on logical reflection based on both first-order formalisms and higher-order logics, including, for example, [75, 16, 37, 5, 49, 62, 60].

An issue that only recently has received attention is axiomatizing the notion of a *reflective logic* in a *logic-independent* way, that is, within a metatheory of general logics [50], so that we can view and compare the different instances of logical reflection based on different formalisms as special cases of a general concept. This has been done by Clavel and Meseguer [22, 17]. We present below the general notion of logic, called an *entailment system* [50], on which their actual definition of reflective logic is based.

3.1 Entailment Systems and Reflective Logics

We assume that logical syntax is given by a *signature* Σ that provides a grammar for building *sentences*. For first-order logic, a typical signature consists of a collection of function and predicate symbols which are used to build up sentences by means of the usual logical connectives. In general, it is enough to assume that for each logic there is a category **Sign** of possible signatures, and a functor *sen* assigning to each signature Σ the set $sen(\Sigma)$ of all its sentences.

For a given signature Σ in **Sign**, *entailment* (also called *provability*) of a sentence $\varphi \in sen(\Sigma)$ from a set of axioms $\Gamma \subseteq sen(\Sigma)$ is a relation $\Gamma \vdash \varphi$ that holds if and only if we can prove φ from the axioms Γ using the rules of the logic. We make this relation relative to a signature. In what follows, $|\mathcal{C}|$ denotes the collection of objects of a category $\mathcal{C}$.

Definition 1. [50] *An* entailment system *is a triple* $\mathcal{E} = (\mathbf{Sign}, sen, \vdash)$ *such that*

- **Sign** *is a category whose objects are called* signatures*,*
- $sen : \mathbf{Sign} \longrightarrow \mathbf{Set}$ *is a functor associating to each signature* Σ *a corresponding set of* Σ-sentences*, and*
- $\vdash$ *is a function associating to each* $\Sigma \in |\mathbf{Sign}|$ *a binary relation* $\vdash_\Sigma \subseteq \mathcal{P}(sen(\Sigma)) \times sen(\Sigma)$ *called* Σ-entailment *such that the following properties are satisfied:*
  1. *reflexivity: for any* $\varphi \in sen(\Sigma)$, $\{\varphi\} \vdash_\Sigma \varphi$,
  2. *monotonicity: if* $\Gamma \vdash_\Sigma \varphi$ *and* $\Gamma' \supseteq \Gamma$ *then* $\Gamma' \vdash_\Sigma \varphi$,
  3. *transitivity: if* $\Gamma \vdash_\Sigma \varphi_i$, *for all* $i \in I$, *and* $\Gamma \cup \{\varphi_i \mid i \in I\} \vdash_\Sigma \psi$, *then* $\Gamma \vdash_\Sigma \psi$,
  4. $\vdash$*-translation: if* $\Gamma \vdash_\Sigma \varphi$, *then for any signature morphism* $H : \Sigma \longrightarrow \Sigma'$ *in* **Sign**, $sen(H)(\Gamma) \vdash_{\Sigma'} sen(H)(\varphi)$, *where* $sen(H)(\Gamma) = \{sen(H)(\varphi) \mid \varphi \in \Gamma\}$, *as is standard.*

Given an entailment system $\mathcal{E}$, its category **Th** of *theories*[1] has as objects pairs $T = (\Sigma, \Gamma)$ with Σ a signature and $\Gamma \subseteq sen(\Sigma)$. A *theory morphism* (also called a *theory interpretation*) $H : (\Sigma, \Gamma) \longrightarrow (\Sigma', \Gamma')$ is a signature morphism $H : \Sigma \longrightarrow \Sigma'$ such that if $\varphi \in \Gamma$, then $\Gamma' \vdash_{\Sigma'} sen(H)(\varphi)$. By composing with the forgetful functor $sign : \mathbf{Th} \longrightarrow \mathbf{Sign}$, with $sign(\Sigma, \Gamma) = \Sigma$, we can extend the functor $sen : \mathbf{Sign} \longrightarrow \mathbf{Set}$ to a functor $sen : \mathbf{Th} \longrightarrow \mathbf{Set}$, i.e., we define $sen(T) = sen(sign(T))$.

Reflection can now be defined as a property of an entailment system. Although stronger requirements can be given (see [22, 17, 11]) the most basic property one wants is the capacity to metarepresent *theories and sentences* as expressions at the object level, and to then *simulate deduction* in those theories using the corresponding metarepresentations. This is captured by the notion of a *universal theory* defined below.

Definition 2. *([22, 17]) Given an entailment system* $\mathcal{E}$ *and a nonempty set of theories* $\mathcal{C}$ *in it, a theory* U *is* $\mathcal{C}$-universal *if there is a function, called a* representation function,

$$\overline{(\_ \vdash \_)} : \bigcup_{T \in \mathcal{C}} (\{T\} \times sen(T)) \longrightarrow sen(U)\,,$$

[1] What we call theories are sometimes called *theory presentations* in the literature.

such that for each $T \in \mathcal{C}, \varphi \in sen(T)$,

$$T \vdash \varphi \;\; \textit{iff} \;\; U \vdash \overline{T \vdash \varphi}\,.$$

If, in addition, $U \in \mathcal{C}$, then the entailment system $\mathcal{E}$ is called $\mathcal{C}$-reflective. *Finally, a* reflective logic *is a logic whose entailment system is $\mathcal{C}$-reflective for $\mathcal{C}$, the class of all finitely presentable theories in the logic.*

3.2 Reflection in Rewriting Logic and Maude

Rewriting logic is reflective in the precise axiomatic sense of Definition 2 above [17, 23]. This is particularly useful for our purposes here, not only because of the advantages that this provides in a logically reflective language design such as Maude, but also because of the powerful ways in which, as explained in Section 4, logical reflection and distributed object-oriented reflection can be combined.

Indeed, as required by Definition 2, rewriting logic has a universal theory $\mathcal{U}$ and a representation function $\overline{(\_ \vdash \_)}$ encoding pairs consisting of a rewrite theory $\mathcal{R}$ and a sentence in it as sentences in $\mathcal{U}$. Specifically, for any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) and any terms t, t' in $\mathcal{R}$, the representation function is defined by

$$\overline{\mathcal{R} \vdash t \longrightarrow t'} = \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t}' \rangle\,,$$

where $\overline{\mathcal{R}}$, $\overline{t}$, and $\overline{t}'$ are ground terms in $\mathcal{U}$.

Since $\mathcal{U}$ is representable in itself, we can achieve a "reflective tower" with an arbitrary number of levels of reflection, since we have

$$\mathcal{R} \vdash t \rightarrow t' \;\Leftrightarrow\; \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \rightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle \;\Leftrightarrow\; \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \rangle \rightarrow \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \rangle \ldots$$

Reflection is systematically exploited in the Maude rewriting logic language implementation [19, 18], that provides key features of the universal theory $\mathcal{U}$ in a built-in module called `META-LEVEL`. In particular, `META-LEVEL` has sorts `Term` and `Module`, so that the representations $\overline{t}$ and $\overline{\mathcal{R}}$ of a term t and a module $\mathcal{R}$ have sorts `Term` and `Module`, respectively. `META-LEVEL` has also functions `meta-reduce`$(\overline{\mathcal{R}}, \overline{t})$, `meta-rewrite`$(\overline{\mathcal{R}}, \overline{t}, n)$, and `meta-apply`$(\overline{\mathcal{R}}, \overline{t}, \overline{l}, \overline{\sigma}, n)$ which return, respectively, the representation of the reduced form of a term t using the equations in the module $\mathcal{R}$, the representation of the result of rewriting a term t at most n steps with the default interpreter using the rules in the module $\mathcal{R}$, and the (representation of the) result of applying a rule labeled l in the module $\mathcal{R}$ to a term t at the top with the $(n + 1)$th match consistent with the partial substitution σ. As the universal theory $\mathcal{U}$ that it implements in a built-in fashion, `META-LEVEL` can also support a reflective tower with an arbitrary number of levels of reflection.

4 Semantic Models of Distributed Object Reflection

We first present, in Section 4.1, a "Russian dolls" model of distributed object reflection first sketched in [52]. We then motivate the need for logical reflection in Section 4.2 by explaining how mobility and code adaptation can be naturally supported. Our generic model of Russian dolls with logical reflection is then explained in Section 4.3 and is illustrated by means of the Mobile Maude [27] language design.

4.1 Distributed Object Reflection through Russian Dolls

In simple situations the distributed state of an object-based system can be conceptualized as a *flat configuration* involving objects and messages. As explained in Section 2, we can visualize this configuration as a "soup" in which the objects and messages float and interact with each other through asynchronous message passing and/or other synchronous interactions. In practice, however, there are often good reasons for *having boundaries* that circumscribe parts of a distributed object state. For example, the Internet is not really a flat network, but a *network of networks*, having different network *domains*, that may not be directly accessible except through specific gateways, firewalls, and so on. This means that in general we should not think of a distributed state as a *flat soup*, but as a *soup of soups*, each enclosed within specific boundaries. Of course, *soups can be nested within other soups* with any desirable depth of nesting. This suggests a *Russian dolls* metaphor: the charming Russian folk art dolls that contain inside other dolls, which in turn contain others, and so on.

Mathematically, this nested structuring of the state can be specified by *boundary operators* of the general form,

$$b : s_1 \dots s_n\ \texttt{Configuration} \longrightarrow \texttt{Configuration},$$

where $s_1 \dots s_n$ are additional sorts, called the *parameters* of the boundary operator, that may be needed to endow the configuration wrapped by b with additional information. The simplest possible example is a boundary operator of the form,

$$\{\_\} : \texttt{Configuration} \longrightarrow \texttt{Configuration},$$

which just wraps a configuration adding no extra information. A more flexible, yet still simple, variant is provided by an operator

$$\{\_ \mid \_\} : \texttt{Location}\ \texttt{Configuration} \longrightarrow \texttt{Configuration},$$

where each wrapped configuration is now *located* in a specific location l of sort `Location`. The set of locations may then have additional structure such as, for example, being a free monoid. Another variant is an operator that adds an interface to a configuration of objects and messages

$$\{\_ \mid \_\} : \texttt{Interface}\ \texttt{Configuration} \longrightarrow \texttt{Configuration}.$$

An interface can be as simple as a pair of sets of object identifiers (ρ, χ) where ρ is a subset of identifiers of objects in the configuration called the *receptionists* and χ is the set of identifiers of objects external to the configuration but accessible from within the configuration. Only receptionists are visible from outside the wrapped configuration. An interface could be more complex, for example specifying the types of messages that can be received or sent.

Yet another quite general method for defining boundary operators is by means of *object classes with a configuration-valued attribute*. These are classes of the general form,

```
class C | conf : Configuration, ATTS .
```

with `ATTS` the remaining attribute declarations for the class. That is, a configuration is now *wrapped inside the state of a containing object*. We call such a containing object a *metaobject*, and the objects in its internal configuration its *subobjects*. Similarly, for other boundary operators b_1, b_2, objects O_1, O_2, remaining configurations C_1, C_2, and parameters $\vec{p}_1, \vec{p}_2$, whenever we have a nested configuration of the form,

$$b_1(\vec{p}_1, O_1 C_1 b_2(\vec{p}_2, O_2 C_2))$$

we call O_1 a *metaobject*, and O_2 is then one of its *subobjects*. Note that the metaobject-subobject relation can be both *many-to-many* and *nested*. For example in the nested configuration

$$b_1(\vec{p}_1, O_1 O_2 C_1 b_2(\vec{p}_2, O_3 O_4 C_2 b_3(\vec{p}_3, O_5 C_3)))$$

the objects O_1, O_2 are both metaobjects of O_3, O_4, which in turn are both metaobjects of O_5. We then say that O_1, O_2 are *meta-metaobjects* of O_5, and that O_5 is one of their *sub-subobjets*, and we can extend this terminology in the obvious way to any number of nesting levels.

Although the Russian dolls model of distributed object reflection is quite simple, it can provide a clear rewriting logic semantics allowing us to express quite sophisticated models of distributed object reflection already proposed in the literature as special instances of the general framework. In Sections 5 and 6 we discuss several such models that we have axiomatized this way. Their semantics is provided by specifying two things:

1. the specific *wrapping and nesting discipline* defining the allowable nested configurations (specified by an *equational theory*, with an associated *signature* of sorts, subsorts, and operators); and
2. the *interaction semantics* between objects and subobjects, which is typically specified by *boundary-crossing rewrite rules*.

By boundary-crossing rewrite rules we mean rewrite rules that allow objects and messages to interact across boundaries. For example, a metaobject O_1 may intercept and encrypt with a function k the contents of a message of the form $(M \rhd O_3)$ sent to an outside object O_3 by its subobject O_2. This could be specified with a rewrite rule of the form,

$$b_1(\vec{p}_1, O_1 C_1 b_2(\vec{p}_2, O_2(M \rhd O_3) C_2)) \to b_1(\vec{p}_1, O_1(k(M) \rhd O_3) C_1 b_2(\vec{p}_2, O_2 C_2))$$

The reflective architecture thus defined is typically *very generic*, allowing the objects involved to belong to a wide range of object classes. Of course, some very general assumptions about object behavior in such classes (for example, communication by asynchronous message passing) may be required for correctness. Therefore, in each concrete instance of a reflective architecture of this kind, besides the generic interaction semantics provided by the boundary-crossing rewrite rules we have also an *application-specific semantics* provided by the rewrite rules for the concrete classes of objects involved in that specific instance. One of the great advantages of reflective architectures is of course the *modularity* and *separation of concerns* that they support, since the reflective interaction semantics is completely independent of the application-specific semantics of each instance.

Reflective architectures based on the Russian dolls paradigm are quite expressive, in the sense that metaobjects can perform a wide range of services and can control their subobjects in many different ways. For example, they can:

- provide security, fault-tolerance, and other communication service features (for example, different forms of quality-of-service properties) in a modular way;
- broadcast information to their subobjects, and gather and aggregate their responses;
- perform system monitoring in a distributed and hierarchical way, and take appropriate metalevel actions (for example, anomaly detection and response);
- control the execution of their underlying subobjects in different ways, including freezing,[2] unfreezing, and scheduling of subobjects;
- create new subobjects or delete existing ones.

However, the Russian dolls model of reflection needs to be generalized in order to provide more powerful features for *code morphing*, *runtime dynamic adaptation*, and *mobility*. Models for such features require the use of *logical reflection*, as explained below.

4.2 Metaobjects with Logical Reflection

We can motivate the need for logical reflection by considering mobility. If objects don't move, their *code* can be static: it can typically be compiled once and for all

[2] Freezing of the state of subobjects can be specified by applying *frozen* operators, that forbid rewriting in any of their arguments, or by other type-theoretic means. Maude 2.0 supports the declaration of frozen operators with the `frozen` attribute.

to run on the execution engine available at the specific location where the object resides. By contrast, a *mobile object* has to carry with it its own code, which can then be executed in the different remote locations visited by the object. Code mobility is needed because the execution engines in those remote locations typically *have no a priori knowledge* of the mobile object's code.

How can this be formally specified in rewriting logic, and how can it be executed in a language like Maude? The key idea is to use logical reflection, and to regard mobile objects as *metaobjects whose code and state are metarepresented as values of some of their attributes.* We can for example define a class,

```
class Mobile | s : Term, mod : Module, ATTS .
```

where we assume that the module defining the class `Mobile` of mobile objects imports the `META-LEVEL` module. Then the value of the `mod` attribute is a term of sort `Module` in `META-LEVEL`, namely the meta-representation $\overline{\mathcal{R}}$ of the rewrite theory $\mathcal{R}$ containing the rules of the mobile object. Similarly the value of the `s` attribute is a term $\overline{C}$ of sort `Term` in `META-LEVEL`, namely the metarepresentation of a configuration C corresponding to the current state of the mobile object. In Mobile Maude [27] the configuration C contains an object *having the same name as that of its enclosing metaobject*, plus current incoming and outgoing messages; that its, the mobile metaobject contains the metarepresentation of an *homunculus subobject* with its same name.

How can mobile objects then be executed in different remote locations without prior knowledge of their code? The idea is that the execution engine of each remote location only needs to know about `META-LEVEL` and about the rewrite rules of the `Mobile` class; they need no knowledge whatever of the mobile object's code, that is, of the particular application-specific rewrite theory $\mathcal{R}$ specifying that code. We can illustrate this idea with the following key rule for the `Mobile` class [27]:

```
rl [do-something] : < M : Mobile | s : T, mod : MOD > =>
     < M : Mobile | s : meta-rewrite(MOD,T,1), mod : MOD > .
```

where, as explained in Section 3.2, `meta-rewrite(MOD,T,1)` is a `META-LEVEL` expression that performs at the metalevel one rewrite step of the term metarepresented by `T` with the rewrite rules of the module metarepresented by `MOD`. One can easily imagine extra attributes `ATTS` for such mobile objects, as suggested above. For example, to prevent *runaway mobile objects* one may wish to have a `gas` attribute, indicating the overall amount of computational resources that the object will be allowed to use in a given location; then each application of the `do-something` rule could decrease the `gas` amount by one unit.

Another way of motivating the need for metaobjects that make an essential use of logical reflection is by considering *dynamic code adaptation.* Consider a class of adaptive objects which can change their code at runtime to deal with new situations. They may for example be mobile objects that—depending on the resources available at different physical locations and on other considerations such as security warnings or attacks—can change the protocols that they use to communicate with other objects. Furthermore, those changes need not be restricted to a fixed, finite repertoire of choices; they may be much more finely tunable by means of different *parameters*, allowing a possibly infinite range of code specializations. Again, the question is how to formally specify metaobjects that can adapt in this way. The answer is similar to the case of mobile objects, namely, such metaobjects should have their code and their state metarepresented. We can define a class, say,

```
class Adaptive | s : Term, mod : Module, ATTS .
```

where we assume that the module defining the class `Adaptive` of adaptive objects imports the `META-LEVEL` module. Therefore, as before, the value of the `mod` attribute will be a term of sort `Module` in `META-LEVEL`, namely the metarepresentation $\overline{\mathcal{R}}$ of the rewrite theory $\mathcal{R}$ containing the current rules of the adaptive object. Similarly the value of the `s` attribute will be a term $\overline{C}$ of sort `Term` in `META-LEVEL`, namely the metarepresentation of a configuration C corresponding to the current state of the adaptive object. Again, we may adopt the convention that the configuration C contains an object *having the same name as that of its enclosing metaobject*, plus current incoming and outgoing messages; that is, that the adaptive metaobject contains the metarepresentation of an *homunculus subobject* with its same name.

Adaptation may in fact occur along different dimensions and for different purposes. There may be a set of *adaptation policies* dictating which forms of adaptation are suitable at a given moment and with which parameters. Such policies may be implemented by invoking specific *adaptation functions*, which change the code and perhaps also the state representation. These adaptation functions have the general form,

$$a.mod : \texttt{Module}\ s_1 \ldots s_n \longrightarrow \texttt{Module},$$

$$a.s : \texttt{Term Module}\ s_1 \ldots s_n \longrightarrow \texttt{Term},$$

where $s_1 \ldots s_n$ are additional sorts used as *parameters* by the adaptation function. That is, *a.mod* takes (the metarepresentation of) a module and a list of parameters, and returns (the metarepresentation of) a transformed module as a result. Similarly, *a.s* takes (the metarepresentations of) a state and its corresponding module, plus a list of parameters, and returns (the metarepresentation of) a transformed state as a result. A particular policy may dictate that, under certain conditions, these functions are invoked on both the code and the state to adapt the object. For example, assuming that under the given policies `a.mod-i` and `a.s-i` are the i^{th} pair of adaptation functions which should be invoked if a certain condition cond occurs, we can specify that particular adaptation by a rewrite rule of the general schematic form,

```
rl [a-i] : < O : Adaptive | s : T, mod : MOD, ATTS > =>
   < O : Adaptive | s : a.s-i(T,MOD,P), mod : a.mod-i(MOD,P), ATTS' >
                                                        if cond(X) .
```

where ATTS and ATTS' may be entire terms, and not just variables, and where P and X denote lists of variables.

The need for *several levels of logical reflection* may be illustrated by considering the case of objects that are *both mobile and adaptive*. The point is that mobility is a much more generic capability—applicable to object systems of many different classes with very few restrictions—than adaptation. Therefore, it seems reasonable to assume that a mobile language implementation will have the *generic code* supporting the execution and interaction of general mobile objects available in all execution engines at all locations. By contrast, adaptation is much more *application-specific*: it is not reasonable at all to assume a single class of rewrite rules for adaptation that would fit all objects and would be available everywhere. There may be many different classes of adaptive objects for different applications. This means that if an object is both mobile and adaptive, *the adaptation code must move with the object.*

How can this be done? By using *two levels of logical reflection*. There is a meta-metaobject that is a mobile object, say of class Mobile. The value of the mod attribute is now the metarepresentation $\overline{\mathcal{A}}$ of a rewrite theory $\mathcal{A}$ specifying a class, say, Adapt-app of adaptive objects for application app. Similarly, the value of the s attribute will now be the metarepresentation $\overline{C}$ of configuration C containing an *homunculus subobject* with the same name *which is itself an adaptive metaobject*. Therefore, such an homunculus subobject contains in its s attribute the metarepresentation $\overline{C'}$ of another configuration C' containing *another homunculus subobject*, namely the baselevel object that is both mobile and adaptive. We can visualize the two levels of logical reflection involved with the mobile meta-metaobject, the adaptive metaobject, and the baselevel object by considering that the mobile object state must have the form:

$$\langle O : \texttt{Mobile} \mid s : \overline{C_1\langle O : \texttt{Adapt}-\texttt{app} \mid s : \overline{C_2\langle O : Cl \mid ATTS\rangle}, ATT'\rangle, ATT''}\rangle ATT'''\rangle$$

4.3 The General Case: Russian Dolls with Logical Reflection

Having metaobjects with logical reflection allows us to do all that metaobjects without such reflection can do in the Russians dolls model presented in Section 4.1 and more. In particular, mobility and adaptation can be naturally modeled by logical reflection.

The two ideas of Russian dolls and logical reflection can be naturally combined and generalized into a model of *Russian dolls with logical reflection*. The idea is that the distributed state will still be a *nested and distributed soup of soups*—the Russian dolls—but now *the subobjects of a metaobject may or may not be metarepresented*, depending on the specific reflective needs of the application in question. As before, the metaobject-subobject relation can be both *many-to-many* and *nested*. Whether a subobject of a metaobject is metarepresented or not is a *purely local property* in the acyclic graph representing the metaobject-subobject relation. For example, a metaobject O_1 may contain a subobject O_2 that is metarepresented, but O_2 may itself be a metaobject containing a subobject O_3 which is *not* metarepresented. Of course, from O_1's perspective, *both* O_2 and O_3 are metarepresented; but they are metarepresented *at the same level*, that is, using only one level of reflection.

We can illustrate these ideas by explaining how the general model of Russian dolls with logical reflection becomes instantiated in the case of Mobile Maude [27]. In Mobile Maude the two key entities are *processes* and *mobile objects*. Processes are located computational environments where mobile objects can reside. Mobile objects can move between different processes in different locations, and can communicate asynchronously with each other by means of messages. As already explained, each mobile object contains its own code—that is, a rewrite theory $\mathcal{R}$—metarepresented as a term $\overline{\mathcal{R}}$, as well as its own internal state, also metarepresented as $\overline{C}$, for C a configuration containing the *homonimous homunculus subobject*.

Figure 1 shows several processes in two locations, with mobile object $o3$ moving from one process to another, and with object $o1$ sending a message to $o2$.

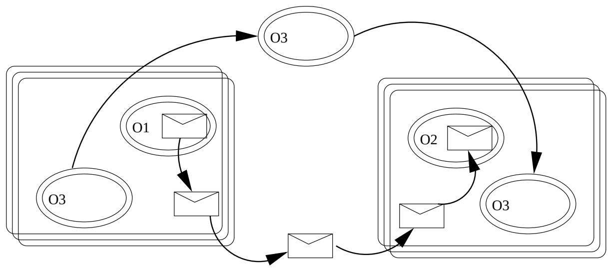

Fig. 1. Object and message mobility

Note the double oval around each mobile object. This is a pictorial way of indicating that mobile objects consist of both the outer metaobject and its inner homonimous homunculus subobject.

From the perspective of our general model of reflection, note that:

- the top-level structure of a Mobile Maude distributed state is a *soup of processes*;
- each process is itself a metaobject, which contains an *inner soup of subobjects*, namely the mobile objects currently residing inside that process; furthermore, those subobjects are *not* metarepresented: since *processes do not move*, there is no need in this case to pay for an extra level of logical reflection;
- however, each mobile object is itself a metaobject whose homonimous homunculus subobject, as well as its rules, *are* metarepresented, since this is needed to support mobility.

The above description only mentions the first three levels of the metaobject-subobject relation. But nothing prevents the homunculus subobject of a mobile object from having other subobjects, and they in turn other subobjects, with no a priori bound on the number of nesting levels. Furthermore, at any level the corresponding subobject may or may not be metarepresented. We have already seen an example of this in Section 4.2, namely an adaptive mobile object whose homunculus subobject is itself an adaptive metaobject which contains another homunculus baseobject, also metarepresented.

5 Some Models of Distributed Reflection

The actor model of computation [35, 10, 1] is a natural model for open distributed systems. To support separation of concerns in developing and (dynamically) adapting open distributed systems, and to support high-level programming abstractions, a number of models of actor reflection have been proposed (see Section 1.1). In this section we show how two of these models, the onion skin [3, 26] and the two-level actor machine (TLAM) [72, 68], can be formalized using the ideas proposed in Section 4.

We will use the classic *ticker* example augmented with a meta-level monitor to illustrate how the two models work. A ticker has a counter which is incremented in response to a `tick` message. A ticker also replies to `time` requests.

```
Vars: c,t in Oid,  n in Nat
Messages:  tick, time@c, reply(n)
Rules:
    < t : Ticker | ctr: n > t <- tick
      ==>
    < t : Ticker | ctr: n+1 > t <- tick

    < t : Ticker | ctr: n > t <- time@c
      ==>
    < t : Ticker | ctr: n > c <- reply(n)
```

Here is a simple computation starting with a ticker and an object `c` that knows the ticker. Concurrently, the ticker processes a `tick` message, and `c` generates a `time` request. Then the ticker processes the request. We annotate the transition arrows with information describing the associated event.

```
 < t : Ticker | ctr: 1 > t <- tick  < c : C | ... t ... >
    = deliver(t<-tick) | exe(c)=>
 < t : Ticker | ctr: 2 > t <- tick t <- time@c < c : C | ... t ... >
    = deliver(time@c) =>
 < t : Ticker | ctr: 2 > t <- tick c <- reply(2) < c : C | ... t ... >
```

Ticker Monitor Specification. A ticker monitor is a metaactor that observes delivery of `time` requests to a ticker and reports the current counter, the number of requests, and the reply address to its coordinator `mc`. The monitor can also be asked to reset the ticker.

```
Vars: c,mc,t,tm in Oids,  n,m in Nat
Messages: log(t,n,m,c), reset, reset-ack
Rules:
[log]
 < tm : Monitor | state: M(t,mc,m) >
   = [deliver(<t:Ticker|ctr:n>t<-time@c)/ ] =>
 < tm : Monitor | state: M(t,mc,m+1) > mc <- log(t,n,m+1,c)
[reset]
 < tm : Monitor | state: M(t,mc,m) > tm <- reset
   =[ /t:= ctr:0]=>
 < tm : Monitor | state: M(t,mc,0) > mc <- reset-ack
```

The metaactor rules are annotated with a pair `[event/effect]`. If `event` is nonempty it specifies the baselevel events that fire the rule. If `effect` is nonempty it specifies an update to the baselevel state. Thus the `log` rule fires when a baselevel transition delivers a `time` message to the monitored ticker. A metalevel message is sent, but there is no effect on the baselevel state. The effect part of the `reset` rule causes the ticker `ctr` attribute to be set to `0` when a `reset` message is delivered. If we add a monitor to the ticker configuration, and rerun the computation scenario we see that the first step is unchanged, but when the `time` request is delivered, the `log` rule of the monitor fires, and a `log` message is sent. We also add a `tm <- reset` message to the initial configuration to see the effect action.

```
 < tm : Monitor | state: M(t,mc,0) >  tm <- reset
 < t : Ticker | ctr: 1 > t <- tick  < c : C | ... t ... >
    = deliver(t<-tick) | ..c.. =>
 < tm : Monitor | state: M(t,mc,0) >  tm <- reset
 < t : Ticker | ctr: 2 > t <- tick t <- time@c < c : C | ... t ... >
    = deliver(t<-time@c) =>
 < tm : Monitor | state: M(t,mc,1) > mc <- log(t,2,1,c)  tm <- reset
 < t : Ticker | ctr: 2 > t <- tick c <- reply(2) < c : C | ... t ... >
```

```
    = deliver(tm<-reset) =>
  < tm : Monitor | state: M(t,mc,1) > mc <- log(t,2,1,c)  mc <- reset-ack
  < t : Ticker | ctr: 0 > t <- tick c <- reply(2) < c : C | ... t ... >
```

We will show how monitor semantics is modeled in two reflective models by desugaring the rule annotations and defining appropriate compositions of monitor with ticker.

5.1 The Onion Skin Model

In the onion skin model each actor has a metaactor that defines the semantics of its primitive actions. For example, a message send by the actor becomes a request to its metaactor to transmit the message. Dually, messages sent to the actor are first received by its metaactor, giving the metaactor the capability to control the receive semantics. If no explicit metaactor behavior is defined, the underlying system semantics provides a default metaactor behavior. An actor composed with its metaactor appears, from the outside, like a normal actor, and thus can be controlled by a further metalevel actor. This gives rise to layers of metalevels and hence the "onion skin" analogy. Distributed services for a group of actors can be expressed in terms of metaactor layers that coordinate to achieve some overall property.

The formalization of the onion skin model presented here is adapted from [26]. Onion skin layers are formalized using *tower objects* that implement individual layers of a tower of objects. Objects at the bottom of a tower represent application-level objects. Each metaobject has as a subobject the object tower immediately below. A top-level object implements the default metaactor and enables communication with the environment.

More precisely there are three object classes that structure towers: `Top`, `MetaTower`, and `Tower`. The class `Tower` has two attributes: `in` and `out`. The attribute `in` is a list of messages representing messages to be delivered to the object. The attribute `out` is a list of outgoing requests for message transmittal and object creation. Baseobjects have the form

```
    < o : BTC |  in: msgs, out: reqs, atts >
```

where `BTC` is a subclass of `Tower`, `msgs` is a list of messages, `reqs` is a list of requests (for creation of objects and transmission of messages) and `atts` stands for the additional internal state attributes. Baselevel rules have the form

```
   < o : BTC |  in: msgs, out: reqs, atts >
     =>
   < o : BTC |  in: msgs', out: reqs . reqs', atts >
```

where `msgs'` is a tail of `msgs` (the result of removing zero or more messages from the beginning), and `reqs . reqs'` is the concatenation of the original outgoing requests with newly generated requests, `reqs'`.

The class `MetaTower` is a subclass of the class `Tower`. It has an additional attribute, `base`, whose value is a tower object. The relation between a metaobject and its subobject is the same at each level of the tower—the base-meta relation, hence the name `base` for the attribute. `MetaTower` objects have the form

```
< o : MTC | in: msgs, out: reqs,  base: tobj, atts >
```

where `MTC` is a subclass of `MetaTower`, `tobj` is a tower object and `atts` holds additional internal state attributes. Metalevel rules have the form

```
< o : MTC | in: msgs, out: reqs,
      base: < o : TC | in: dmsgs, out: ureqs, tatts >,
      atts>
  =>
< o : MTC | in: msgs', out: reqs . reqs',
      base: < o : TC | in: dmsgs . dmsgs', out: ureqs', tatts >,
      atts' >
```

where `msgs'` is a tail of `msgs`, `ureqs'` is a tail of `ureqs`, `dmsgs'` is a list of messages, and `reqs'` is a list of requests. The idea is that a metaobject can take a message from its `in` queue, or take a request from the `out` queue of its `base` subobject; and it can place requests in its own `out` queue or place messages in the `in` queue of its `base` subobject.

The class `Top` has one attribute, `base`, whose value is a tower object. Top objects only have rules for communication with the environment. These rules formalize the *default metaactor* behavior.

```
[in]
  o <- msg
  < o : Top | base: < o : TC | in: msgs, out: reqs,  tatts > >
     =>
  < o : Top | base: < o : TC | in: msgs . msg, out: reqs, tatts > >

[out]
  < o : Top | base: < o : TC | in: msgs, out: req . reqs, tatts > >
    =>
  < o : Top | base: < o : TC | in: msgs, out: reqs, tatts > >
  createConf(req)
```

where `createConf(req)` is the configuration whose creation is requested by `req`.

Figure 2 shows a two-level tower, with a message `M` coming in and being modified by the meta-level before delivering it to the base layer. There is also a

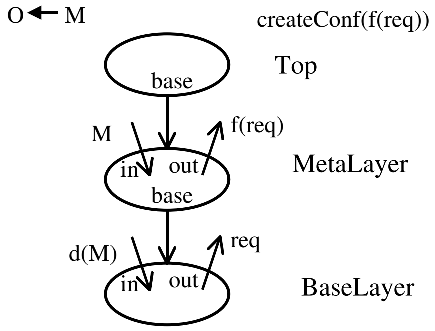

Fig. 2. A metaobject tower

request `req` going up from the base layer, being transformed by the meta layer and the processed by the top layer using the default semantics.

As a concrete illustration we model the ticker with monitor example as a tower. We apply a uniform transformation that maps actor objects to tower baseobjects. (This transformation is defined and proved to preserve semantics in [26].) The transformation maps the `Ticker` class to the `TTicker` class by adding the required `in` and `out` attributes. The ticker rules are then adapted to use these attributes as follows.

```
TowerTicker Rules:
    < t : TTicker | in: tick . msgs, out: reqs,  ctr: n >
      ==>
    < t : TTicker | in: msgs, out: reqs . (t <- tick),  ctr: n+1 >

    < t : TTicker | in: time@c . msgs, out: reqs,  ctr: n >
      ==>
    < t : TTicker | in: msgs, out: reqs . (c <- reply(n)),  ctr: n >
```

The ticker monitor is modeled using a subclass `TMonitor` of `MetaTower` obtained by adding `in`, `out`, and `base` attributes. The monitor rules are adapted as follows. The monitor forwards ticker messages to its ticker subobject and responds directly to `reset` messages. It also transmits all requests from the ticker, adding an appropriate `log` message in the case of a `reply` message.

```
 < t : TMonitor | in: msg . mmsgs, out: mreqs,
                  base: < t : Ticker | in: msgs, out: reqs, ctr: n >
                  state: M(t,mc,m) >
```

```
=>
< t : TMonitor | in: mmsgs, out: mreqs,
                base: < t : Ticker | in: msgs . msg, out: reqs, ctr: n >
                state: M(t,mc,m) >
 if msg in (tick, time@c)

< t : TMonitor | in: reset . mmsgs, out: mreqs,
                base: < t : Ticker | in: msgs, out: reqs, ctr: n >
                state: M(t,mc,m) >
=>
< t : TMonitor | in: mmsgs, out: mreqs . (mc<-reset-ack),
                base: < t : Ticker | in: msgs, out: reqs, ctr: 0 >
                state: M(t,mc,0) >

< t : TMonitor | in: mmsgs, out: mreqs,
                base: < t : Ticker | in: msgs, out: req . reqs, ctr: n>
                state: M(t,mc,m) >
=>
< t : TMonitor | in: mmsgs, out: mreqs . req . qreq
                base: < t : Ticker | in: msgs, out: reqs, ctr: n>
                state: M(t,mc,m') >
 where
   if req := (c <- reply(n'))
   then qreq =  (mc<-log(t,n',m+1,c)) and m' = m+1
   else qreq = mt and m' = m
```

A case study using the tower model to compose an encryption service and a client-server application in which the server may create helper objects and delegate requests to them can be found in [26].

5.2 The Two-Level Actor Machine Model

In the TLAM, a system is composed of two kinds of actors, baseactors and metaactors, distributed over a network of processing nodes. *Baseactors* carry out application level computation, while *metaactors* are part of the runtime system which manages system resources and controls the runtime behavior of the baselevel. Metaactors communicate with each other via message passing as do baselevel actors. Metaactors may also examine and modify the state of the baseactors located on the same node. Baselevel actors and messages have associated runtime annotations that can be set and read by metaactors, but are invisible to baselevel computation. Actions which result in a change of baselevel state are called events. Metaactors may react to events occurring in their node. A TLAM system configuration is a soup of nodes and messages with base and metaactors forming subsoups inside the nodes. A node transition is either a *communication transition* or an *execution transition.* A communication transition moves a message to actors on other nodes from the node's buffer to the external soup or moves a message to an actor on the node from the external soup into the node's mail buffer. An execution transition first applies either a base or metalevel transition rule. If there is an associated baselevel event, then each event-handling rule that matches the event must be applied (in an unspecified order) and the associated annotation-update applied to the new local baselevel configuration and messages.

Formally a TLAM is represented using a reflective object theory. Actors are modeled as objects with identity and state, TLAM configurations as soups of actors and messages, and actor behavior rules as rewrite rules. The trick is to correctly model the metalevel observation of baselevel events and to ensure that the event handling rules are applied when required. One way to do this is to use metalevel strategies to describe the allowed computations. Another way is to wrap the base and metaactor configurations of a node in operators that control when rules can be applied. This works, but the resulting model is not so natural. Here we show how the idea of Russian dolls with logical reflection allows a quite natural modeling of the basic TLAM, an in fact provides a basis for natural extensions to model scheduling, communication protocols and other concerns.

A baselevel actor system is represented quite directly as a special case of object module configurations. A baselevel actor is represented as an object in some baseactor class `BC`. Messages have the form `ba <- v`, where `ba` is the identifier of the intended recipient and `v` is the message contents. Baselevel actor rules are constrained to have the form

```
< ba : BC | atts > [ ba <- v ] => < ba : BC' | atts' > bconf
 if  cond
```

where `BC` and `BC'` are baselevel classes, `atts` and `atts'` are attribute-value sets appropriate for the corresponding classes, `bconf` is a configuration of newly created baseactors and messages, and `[ ]` indicates that the message part is optional.

A metaactor is an object in some metaactor class `MC` and metamessages have the form `ma <- mv` where `ma` is the identifier of a metaactor. Metaactors are grouped as subobject configurations of nodes, which in turn are metametaobjects. A *node* is an object of class `TLAM-Node`. A node has an attribute `conf` whose value is a configuration containing metaactors and messages. Each node has two special metaactors, one of class `BMC` that represents and controls the baselevel execution behavior, and the other of class `CMC` that represents and controls the baselevel communication semantics. These metaactors also implement the event handling semantics associated with baselevel events. Thus a node has the form

```
<nu : TLAM-Node | conf: <bma : BMC | ...> <cma : CMC | ...> mconf>
```

A metaobject of class `BMC` has attributes `bMod`, `bConf`, `eReg`, `pendEv`, and `waitFor`.

```
<bma : BMC | bMod: !BM, bConf: !bc, eReg: emap, pendEv: ev, waitFor: W>
```

The value of the attribute `bMod` is the metarepresentation of the baselevel module `BM`, and the value of the attribute `bConf` is the metarepresentation of the baselevel configuration `bc`. We use the convention that if `foo` denotes a baselevel entity then `!foo` denotes the metarepresentation of that entity. Thus `!bc` is the metarepresentation of `bc`. (This is the teletype analog of the overbar notation.)

The remaining attributes deal with execution event handling. An execution event is the modification of a node's baselevel actor configuration, resulting from applying either a base or a metalevel rule. An execution event is represented by a term of the form `exe(!bconf, !update, cmap)` where `bconf` is the initial baselevel configuration and `update` is a baselevel configuration that specifies new states for some existing baselevel actors, as well as newly created baselevel actors and messages. `cmap` maps newly created baselevel actors and messages to identifiers of existing baselevel actors on whose behalf the new elements are being created. This is used to maintain a model of the baselevel causality and acquaintance relations.

Event handling is modeled by sending notifications to metaactors registered for the event. A metaactor is registered for an event only if it has an event-handling rule matching this event that generates a reply to the notifying actor, in this case the behavior metaactor. The value `emap` of the attribute `eReg` maps an execution event to the set of names of metaactors that are registered to be notified about the event. The mapping is computed using the term `apply(emap,event)`. The value `ev` of `pendEv` is either an execution event or the special constant `none`, indicating that no event handling is in progress. The value `W` of `waitFor` is the set of identifiers of metaactors for which a notification reply has not been received.

A communication metaactor (class `CMC`) has attributes `sendQ`, `arriveQ`, `eReg`, `pendEv`, and `waitFor`.

```
< cma : CMC | sendQ: smsgs, arriveQ: amsgs, eReg: emap,
              pendMsg: ev,  waitFor: W >
```

The value `smsgs` of `sendQ` is a list of pairs of the form `(!msg,!sndr)`, representing requests from baselevel actor `sndr` to send `msg`. The value `amsgs` of `arriveQ` is a list of message arrivals, triples of the form `(!msg,!sndr,amap)` where `amap` is the message annotation map. The remaining attributes are event handling attributes analogous to those of the class `BMC`. A message arrival event has the form `arrive(!(ba <- v), !sndr, amap)` indicating the arrival at the node of a message for `ba`, sent by `sndr` and having annotations `amap`. A message send event has the form `send(!(ba <- v), !sndr, amap)` indicating that a message having annotations `amap` is to be sent to `ba`, by `sndr`. [3]

[3] We keep annotations at the metalevel. In the case of baseactor annotations we let the metaactors maintain them internally. Thus sharing of annotations must be by communication between metaactors. Shared annotations could be modeled as an additional attribute of the class `BMC`, but it seems cleaner to eliminate shared data.

In the TLAM *metalevel transition rules* have the form

$$mactor[mmsg] \xrightarrow[update]{baconf} mactor' \ mconf \ \ \textbf{if} \ \varphi$$

where *mactor* is a metaactor, $[mmsg]$ is an optional metalevel message addressed to the actor, $mactor'$ is the same metaactor with its state possibly modified, *mconf* is a configuration of newly created baselevel actors and messages, and φ is a predicate constraining the actor-message pairs to which the rule applies. *update* is either empty or has the form $(bconf, cmap)$. When the rule is applied, *baconf* is bound to the baselevel actor configuration on the node where the metaactor is located.

Metaactor rules with empty *update* are represented directly as object rewrite rules.

```
 < ma : MC | atts >  [mmsg] => < ma : MC' | atts' >  mconf
   if cond
```

Rules with non-empty update are represented using synchronization with the behavior actor.

```
   < ma : MC | atts > [msg]
   < bma : BMC | bMod: !BM, bConf: !bc, pendEv: none >
   =>
   < ma : MC | atts > [msg]
   < bma : BMC | bMod: !BM, bConf: !bc, pendEv: evt,  started: false >
   if evt := exe(!BM:!bc, !update, cmap)  and cond'

[startMEvent]
   < bma : BMC | eReg: emap, pendEv: evt,  started: false >
   =>
   < bma : BMC | eReg: emap, pendEv: evt, started: true, waitFor: W >
    notifyall(W,evt)
    if  evt =/= none and W = apply(emap, evt)
```

where `notifyall(W,evt)` is the set of messages `ma <- notify(evt)` for `ma` in `W`. The behavior metaactor also manages the scheduling of baselevel transitions. The general rule allows any enabled baselevel rule to be applied.

```
[startExecution]
   < bma : BMC | bMod: !BM, bConf: !bc, pendEv: none >
    =>
   < bma : BMC | bMod: !BM, bConf: !bc, pendEv: evt, started: false >
```

In the case of messages they are components of the metalevel messages that transport the baselevel message. This is consistent with the fact that annotations cannot be seen at the baselevel. By keeping them at the metalevel we avoid the need to extend baselevel modules with annotation data types.

```
   if (!ba, rname, subst) in enabled(!BM, !bc)
      and !update := Update(!BM, !bc, !ba, rname, subst)
      and cmap := makecmap(!update,!ba)
      and evt := exe(!BM:!bc, !update, cmap)
```

The condition

$$(!\mathtt{ba}, \mathtt{rname}, \mathtt{subst})\mathtt{inenabled}(!\mathtt{BM}, !\mathtt{bc})$$

holds if

$$\mathtt{meta}-\mathtt{apply}(!\mathtt{BM}, \mathtt{restrict}(!\mathtt{bc}, !\mathtt{ba}), \mathtt{rname}, \mathtt{subst}, 0)$$

succeeds, where `restrict(!bc,!ba)` is the term metarepresenting the subconfiguration of `bc` containing the actor named `ba` and any pending messages for that actor. `Update(!BM, !bc, !ba, rname, subst)` is the term metarepresenting the result of updating `bc` using the result of `meta-apply` as above. `cmap` maps new actors and messages to the active actor `ba`. When an execution event is being processed, the behavior metaactor waits for acknowledgments from the registered metaactors.

```
[continueExecution]
   < bma : BMC | pendEv: evt,  started: true, waitFor: W ma >
    bma <- notified @ ma
   =>
   < bma : BMC | pendEv: evt,  started: true,  waitFor: W >
   if not(ma in W)
```

When all registered metaactors have acknowledged notification the behavior metaactor completes the execution event by updating its model of the base-level configuration, and transmitting newly sent messages to the communication manager.

```
[completeExecution]
   < bma : BMC | bMod: !BM, bConf: !bc, started: true,
                 pendEv: exe(!BM:!bc, !update, cmap), waitFor: mt >
   < cma : CMC | sendQ: smsgs >
   =>
   < bma : BMC | bMod: !BM, bConf: !bc', pendEv: none  >
   < cma : CMC | sendQ: smsgs . smsgs' >
```

where `bc'` is the result of applying `update` to `!bc`, and `smsgs'` contains elements of the form `(!msg,cmap(!msg))` for each `msg` specified by `update`.

At the node level, baselevel messages are represented by requests to the communications metaactor of the form `cma <- deliver(!msg, !sndr, amap)`. These requests are queued for later processing.

```
[arriveQ]
   < cma : CMC |  arriveQ: amsgs > cma <- deliver(!msg, !sndr, amap)
   =>
   < cma : CMC | arriveQ: amsgs . (!msg, !sndr, amap) >
```

For baselevel messages to be delivered, the communications metaactor first notifies registered metaactors.

```
[startArrival]
   < cma : CMC | arriveQ: (!msg, !sndr, amap) . amsgs,
                 eReg: emap, pendMsg: none >
   =>
   < cma : CMC | arriveQ: amsgs, pendMsg: !msg, waitFor: W >
   notifyall(W, evt)
   if evt := arrive(!msg, !sndr, amap) and  W := apply(emap, evt)
```

The communications metaactor waits for acknowledgments from all notified metaactors and then transmits the message to the behavior metaactor.

```
[continueArrival]
   < cma : CMC | pendMsg: !msg,  waitFor: W ma > cma <- notified(ma)
   =>
   < cma : CMC | pendMsg: !msg,  waitFor: W >  if not(ma in W)

[completeArrival]
   < cma : CMC | pendMsg: !msg,  waitFor: mt >
   < bma : BMC | bMod: !BM, bConf: !bc, pendEv: none >
   =>
   < cma : CMC | pendMsg: none >
   < bma : BMC | bMod: !BM, bConf: deliver(!BM,!bc,!msg) >
```

For each baselevel message to be sent, the communications metaactor notifies metaactors registered for send events, collects annotations, and then sends a `deliver` message to the communications metaactor co-located with the message target.

```
[startSend]
   < cma : CMC | sendQ: (!msg, !sndr) o smsgs, eReg: emap, pendMsg: none >
   =>
   < cma : CMC | sendQ: smsgs, pendMsg: (!msg, !sndr, mt),  waitFor: W >
   notifyall(W, evt)
   if evt := send(!msg, !sndr) and  W := apply(emap, evt)

[continueSend]
   < cma : CMC | pendMsg: (!msg, !sndr, amap), waitFor: W ma >
   cma <- notified(amap', ma)
   =>
   < cma : CMC | pendMsg: (!msg, !sndr, amap . amap'),  waitFor: W >
   if not(ma in W)

[completeSend]
   < cma : CMC | pendMsg: (!msg, !sndr, amap),  waitFor: mt >
   =>
   < cma : CMC | pendMsg: none,  waitFor: mt >
   mailQ(target(!msg)) <- deliver(!msg, !sndr, amap)
```

Notice that the communication metaactor can receive requests for message processing interleaved with the processing of a message event, but may not interleave processing of two or more message events. The two rules that involve synchronization with the behavior metaactor are the mechanism to ensure a consistent causal connection between base and metalevels.

Representing the ticker and monitor example in the TLAM model is now simple. The ticker itself is unchanged. The monitor rules are modified as follows.

```
TLAM Monitor Rules:
  < mt : Monitor | state: M(t,mc,m) >
  mt <- notify(send(!(c<-reply(n)), !t))@cma
  =>
  < mt : Monitor | state: M(t,mc,m+1) >
  cma <- notify-ack(mt) mc <- log(t,n,m+1,c)

  < mt : Monitor | state: M(t,mc,m) >  mt <- reset
  < bma : BMC | bMod: !BM, bConf: !bc, pendEv: none >
  =>
  < mt : Monitor | state: M(t,mc,m+1) >  mc <- reset-ack
  < bma : BMC | bMod: !BM, bConf: !bc, pendEv: evt,  started: false >
   if evt := ev(mt, !BM:!bc, ![t -> ctr -> 0])
```

Extensions of the TLAM model may modify the behavior of `BMC` to introduce scheduling algorithms and an interface to control scheduling. They may modify `CMC` to provide mechanisms for applying communications protocols and establishing policies for which protocols to apply.

6 Towards Semantically-Based Reflective Middleware

As indicated in Section 1.1 there are a number of ongoing efforts to develop reflective middleware systems. In the following we discuss three examples and indicate how they might be given semantics in our framework.

6.1 Anatomy of a Computational Grid

Grid computing addresses the problem of flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources, called *virtual organizations*. Grid programming applications include: multidisciplinary simulation, crisis management, and scientific data analysis. Middleware for Grid computing must provide a range of authentication, authorization, resource access, resource discovery, and collaboration services that cross platform and institution boundaries and allow participants to negotiate, manage, and exploit sharing relationships. In [31] a grid protocol architecture is proposed consisting of several layers as shown in Figure 3. The Grid Fabric layer provides

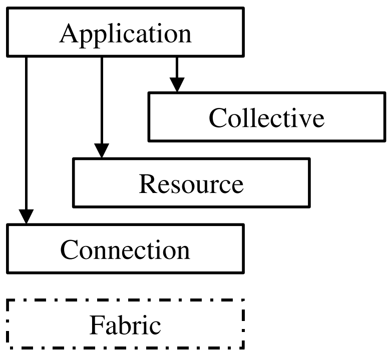

Fig. 3. Grid architecture

the resources to which shared access is mediated by Grid protocols: for example, computational resources, storage systems, catalogs, network resources, and sensors. A "resource" may also be a logical entity, such as a distributed file system, computer cluster, or distributed computer pool. The Connectivity layer defines core communication and authentication protocols required for Grid-specific network transactions. Communication protocols enable the exchange of data between Fabric layer resources. Communication requirements include transport, routing, and naming. Authentication requirements include single sign on, delegation, integration with local security solutions, user-based trust relations. The Resource layer builds on Connectivity layer communication and authentication protocols to define protocols for the secure negotiation, initiation, monitoring, control, accounting, and payment of sharing operations on individual resources. Resource layer protocols are concerned entirely with individual resources. The Collective layer contains protocols and services that are global in nature and capture interactions across collections of resources. Examples include: directory services; co-allocation, scheduling, and brokering services; monitoring and diagnostics services; data replication services; and problem solving environments. The final layer is the Application layer.

This Grid architecture corresponds well to the TLAM model with the application layer corresponding to the baselevel, the collective, resource and communication layers, corresponding to the metalevel (middleware services), and the fabric layer corresponding to the TLAM infrastructure. The further layering of the middleware services is similar in spirit to the TLAM approach of identifying core services and building more complex global services on these foundations [68]. This correspondence to the TLAM suggests that techniques developed to compose and reason about TLAM specifications can be applied to formally specify and reason about grid-based systems.

6.2 Quality Objects

Quality Objects (QuO) [46, 57] is a framework for including Quality of Service (QoS) in distributed object applications. QuO supports the specification of QoS contracts between clients and service providers, runtime monitoring of contracts, and adaptation to changing system conditions. QuO is based on the CORBA middleware standard. The operating regions and service requirements of an application are encoded in contracts, which describe the possible states the system might be in and actions to take when the state changes. QuO inserts delegates in the CORBA functional path to support adaptive behavior upon method call and return. The delegate uses a set of contracts to check the state of the system and choose a behavior based upon it. System condition objects provide interfaces to system resources, mechanisms, and managers. They are used to capture the states of particular resources, mechanisms, or managers that are required by the contracts and to control them as directed by the contracts. QuO also provides a support for interfacing to below-the-ORB QoS mechanisms. Examples of property management below-the-orb are dependability management using replication, and bandwidth management using RSVP. These mechanisms also support intrusion detection systems via monitoring net traffic or resource access. QuO objects can themselves be subject to QoS contracts, thus leading to multiple levels of reflection.

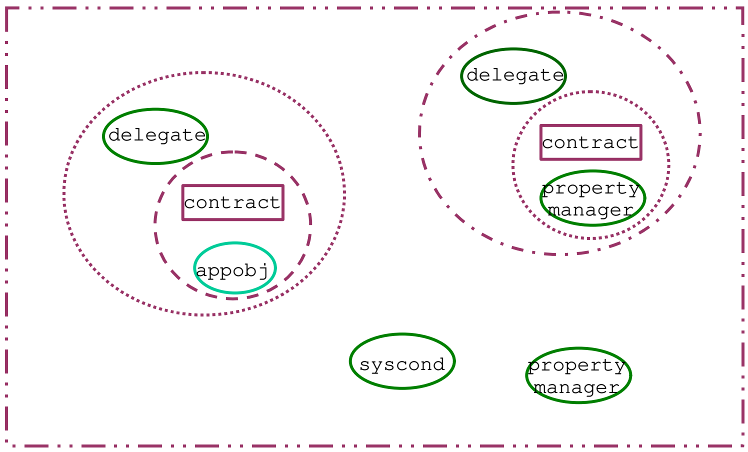

Fig. 4. Quo Architecture

Figure 4 indicates how parts of a QuO system configuration might be mapped to the Russian dolls model. The outer rectangle is a metaobject representing a node. It has a configuration of subobjects including system condition and property manager metaobjects (ovals with solid boundary). The circle on the left represents a delegate metaobject with its application subobject and the applications associated contract. The circle on the right shows the reflective aspect in which a critical property manager is a suboject protected by a contract.

6.3 OpenOrb

The OpenOrb reflective architecture [25, 30] is based on the RM-ODP object model [39] and inspired by the AL-1/D reflective programming language for distributed applications [56]. An RM-ODP object may have multiple interfaces as well as being composed of nested interacting objects. In the OpenOrb architecture every object (interface) then has an associated metaspace supporting inspection and adaptation of the underlying infrastructure for the object. Following AL-1/D, this metaspace is organized as a number of closely related but distinct metaspace models. Currently there are four metamodels: compositional, environment, encapsulation, and resource. The compositional metamodel deals with the way a composite object is composed, i.e., how its components are interconnected, and how these connections are manipulated. There is one compositional metamodel per object. The environmental metamodel is in charge of the environmental computation of an object interface, i.e., how the computation is done. It deals with message arrivals, message selection, dispatching, marshaling, concurrency control, etc. The encapsulation metamodel relates to the set of methods and associated attributes of a particular object interface. Methods scripts can be inspected and changed by accessing this metamodel. Finally, the resource metamodel is concerned with both the resource awareness and resource management of objects in the platform.

The resource model includes abstract resources, resource factories and resource managers. Abstract resources explicitly represent system resources. In addition, there may be various levels of abstractions allowing higher level resources to be constructed on top of lower level resources. Resource managers are responsible for managing resources, that is, such managers either map or multiplex higher level resources onto lower level resources. Virtual task machines (VTMs) are top-level resource abstractions that may encompass several kinds of resources (e.g. CPU, memory and network resources) allocated to a particular task. Resource schedulers are a specialization of managers in charge of managing processing resources such as threads or virtual processors. Resource factories create abstract resources.

Note that objects/interfaces at the metalevel are also open to reflection and have an associated meta-metaspace. This process can be continued, providing a potentially infinite tower of reflection.

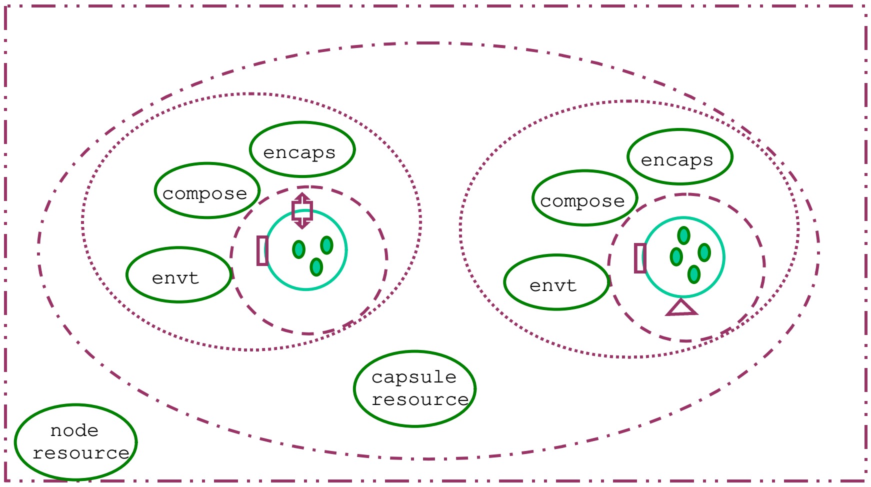

Fig. 5. The OpenOrb Reflective Architecture

Figure 5 illustrates how an OpenOrb system configuration might be represented using the Russian dolls with logical reflection ideas. The outer rectangle is a node metaobject corresponding with its node resource manager. It has a configuration of capsule subobjects, of which the large oval is a representative. The capsule metaobject has a resource manager subobject and several application metaobjects, each containing the metaobjects for the three object specific metamodels, and an ODP object with its multiple interfaces and inner object graph. The shading of the inner objects indicates that these objects are metarepresented subobjects.

7 Concluding Remarks

We have proposed a generic model of distributed object reflection based on rewriting logic The model uses configuration-structuring mechanisms to organize distributed object configurations into hierarchies of metaobjects and subobjects, and logical reflection to treat subobjects as data for increased power, control, and adaptivity. We have also shown how this generic model specializes in natural ways to several well-known models of actor reflection and of reflective middleware.

We envision several directions for future work. One direction is to continue the analysis of reflective middleware to identify different nesting structures, required objects, if any, at each level, and interaction patterns, as expressed by allowed rule structure. One objective is to obtain a deeper understanding of base-meta interactions and principles leading to a useful balance of power and restraint.

A second direction is developing techniques for specification and analysis of reflective distributed systems. Challenges include modeling and reasoning about interactions of different metamodels and services; combining specifications of different aspects/concerns; and modular treatment of different reflective levels.

Acknowledgments. This work was supported in part by National Science Foundation Grants CCR-9900334 and CCR-9900326. The first author's work is supported in part by the ONR MURI Project "A Logical Framework for Adaptive System Interoperability."

The authors would like to thank our collaborators Gul Agha, Manuel Clavel, Grit Denker, Nalini Venkatasubramanian for their many contributions to the work leading to the results presented here. Thanks to Narciso Martí-Oliet both for fruitful collaboration and for help in proofreading. We also thank Doug Schmidt for insightful discussions regarding reflective middleware.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.
2. G. Agha and C. Callsen. Actorspace: An open distributed programming paradigm. In *Proc. ACM Symp. on Principles and Practice of Parallel Programming (PPOPP)*, volume (28:7), pages 23–32, July 1993.
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. G. Agha, S. Frølund, R. Panwar, and D. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *Proceedings of the 3rd IFIP Working Conference on Dependable Computing for Critical Applications*, Sept. 1992.
5. W. E. Aitken, R. L. Constable, and J. L. Underwood. Metalogical frameworks II: Using reflected decision procedures. Technical Report, Computer Sci. Dept., Cornell University, 1993; also, lecture at the Max Planck Institut für Informatik, Saarbrücken, Germany, July 1993.
6. A. Albarrán, F. Durán, and A. Vallecillo. From Maude specifications to SOAP distributed implementations: A smooth transition. In *VI Jornadas Ingenieria del Software y Bases de Datos (JISBD'01)*, pages 419–433, 2001.
7. A. Albarrán, F. Durán, and A. Vallecillo. Maude meets CORBA. In *Second Argentine Symposium on Software Engineering (ASSE'01)*, 2001.
8. J. Andrews. Process-algebraic foundations of separation-of-concerns programming. In *Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Reflection 2001*, Lecture Notes in Computer Science. Springer-Verlag, 2001.
9. M. Astley. *Customization and Composition of Distributed Objects: Policy Management in Distributed Software Architectures.* PhD thesis, University of Illinois, Urbana-Champaign, 1999.
10. H. G. Baker and C. Hewitt. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992. IFIP, Aug. 1977.

11. D. Basin, M. Clavel, and J. Meseguer. Rewriting logic as a metalogical framework. In S. Kapoor and S. Prasad, editors, *FST TCS 2000*, pages 55–80. Springer LNCS, 2000.
12. G. Blair, M. Clarke, F. Costa, G. Coulson, H. Duran, and N. Parlavantzas. The evolution of OpenORB. In Kon and Saikoski [44]. see `http://www.comp.lancs.ac.uk/computing/users/johnstlr/rm2000/`.
13. G. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Middleware '98*, 1998.
14. L. Blair and G. Blair. Composition in multiparadigm specification techniques. In *3rd International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, pages 401–417. Kluwer, 1999.
15. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm`.
16. R. S. Boyer and J. S. Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In R. Boyer and J. Moore, editors, *The Correctness Problem in Computer Science*, pages 103–185. Academic Press, 1981.
17. M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Publications, 2000.
18. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marti-Oliet, and J. Meseguer. Metalevel Computation in Maude. In C. Kirchner and H. Kirchner, editors, *2nd International Workshop on Rewriting Logic and Its Applications, WRLA'98*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`.
19. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. SRI International, January 1999, `http://maude.csl.sri.com`.
20. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. A tutorial on Maude. SRI International, March 2000, `http://maude.csl.sri.com`.
21. M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *Proc. of the CafeOBJ Symposium '98, Numazu, Japan*. CafeOBJ Project, April 1998. `http://maude.csl.sri.com`.
22. M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kiczales, editor, *Proceedings of Reflection'96, San Francisco, California, April 1996*, pages 263–288, 1996. `http://jerry.cs.uiuc.edu/reflection/`.
23. M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. to appear in Theoretical Computer Science, Volume 304, Issues 1-2., 2002.
24. P. Cointe, editor. *Proceedings of Reflection'99*, volume 1616 of *Lecture Notes in Computer Science*. Springer, 1999.
25. F. M. Costa and G. S. Blair. Integrating meta-information management and reflection in middleware. In *Second International Symposium on Distributed Objects and Applications (DOA'00)*, pages 133–143, Sept. 2000.
26. G. Denker, J. Meseguer, and C. L. Talcott. Rewriting semantics of distributed meta objects and composable communication services. In *Third International Workshop on Rewriting Logic and Its Applications (WRLA'2000), Kanazawa, Japan, September 18 — 20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000. `http://www.elsevier.nl/locate/entcs/volume36.html`.

27. F. Durán, S. Eker, P. Lincoln, and J. Meseguer. Principles of Mobile Maude. In *Agent Systems, Mobile Agents, and Applications, ASA/MA 2000*, volume 1882 of *Lecture Notes in Computer Science*, pages 73–85. Springer-Verlag, 2000.
28. F. Durán and A. Vallecillo. Specifying the ODP information viewpoint using Maude. In H. Kilov and K. Baclawski, editors, *Proceedings of Tenth OOPSLA Workshop on Behavioral Semantics*, pages 44–57, October 2001.
29. F. Durán and A. Vallecillo. Writing ODP enterprise specifications in Maude. In J. Cordeiro and H. Kilov, editors, *Proceedings of Workshop On Open Distributed Processing: Enterprise, Computation, Knowledge, Engineering and Realisation (WOODPECKER'01)*, pages 55–68, July 2001.
30. H. A. Duran-Limon and G. S. Blair. The importance of resource management in engineering distributed objects. In *Second International Workshop on Engineering Distributed Objects (EDO2000)*, Nov. 2000.
31. I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 2001.
32. S. Frølund. *Coordinated Distributed Objects: An Actor Based Approach to Synchronization*. MIT Press, 1996.
33. K. Futatsugi and R. Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.
34. A. Grimshaw and W. W. et al. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1), Jan. 1997.
35. C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of 1973 International Joint Conference on Artificial Intelligence*, pages 235–245, Aug. 1973.
36. P. Hill and J. Lloyd. The Gödel language. Technical Report CSTR-92-27, University of Bristol, Computer Science Department, 1992.
37. D. J. Howe. Reflecting the semantics of reflected proof. In P. Aczel, H. Simmons, and S. S. Wainer, editors, *Proof Theory*, pages 229–250. Cambridge University Press, 1990.
38. J.-I. Itoh, Y. Yokote, and R. Lea. Using meta-objects to support optimisation in the apertos operating system. In *USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 147–158. USENIX Association, 1995.
39. ITU-T/ISO. Reference model for open distributed processing. Technical Report ITU-T X.901-X.904 — ISO/IEC IS 10746-(1,2,3), ITU-T/ISO, 1995.
40. G. Kiczales, editor. *Reflection'96*, 1996.
41. G. Kiczales, J. des Riviers, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
42. 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. European Conference on Object-Oriented Programming (ECOOP'97), Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, 1997.
43. F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhães, and R. H. Campbell. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 121–143, New York, April 2000. Springer-Verlag.
44. F. Kon and K. B. Saikoski, editors. *IFIP/ACM Middleware'2000 Workshop on Reflective Middleware*, New York, April 2000. Gordon Blair and Roy Campbell (co-chairs). see `http://www.comp.lancs.ac.uk/computing/users/johnstlr/rm2000/`.

45. F. Kon, A. Singhai, R. H. Campbell, D. Carvalho, R. Moore, and F. J. Ballesteros. 2K: A Reflective, Component-Based Operating System for Rapidly Changing Environments. In *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, July 1998.
46. J. Loyall, R. Schantz, J. Zinky, and D. Bakken. Specifying and measuring quality of service in distributed object systems. In *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98)*, Apr. 1998.
47. P. Maes. Concepts and experiments in computational reflection. In *OOPSLA*, pages 147–155, 1987. in ACM SIGPLAN Notices 22(12).
48. E. Marsden, J. C. R. García, and J.-C. Fabre. Towards validating reflective architectures: formalization of a metaobject protocol. In Kon and Saikoski [44]. see `http://www.comp.lancs.ac.uk/computing/users/johnstlr/rm2000/`.
49. S. Matthews. Reflection in logical systems. In *IMSA'92*, pages 178–183. Information-Technology Promotion Agency, Japan, 1992.
50. J. Meseguer. General logics. In H.-D. E. et al., editor, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.
51. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
52. 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.
53. T. A. Mogensen. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2(3):345–364, 1992.
54. E. Najm and J.-B. Stefani. Computational models for open distributed systems. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-based Distributed Systems, Volume 2*, pages 157–176. Chapman & Hall, 1997.
55. Object Management Group. The Common Object Request Broker: Architecture and Specification, 2.3 ed., June 1999.
56. H. Okamura, Y. Ishikawa, and M. Tokoro. Al-1/d: A distributed programming system with multi-model reflection framework. In A. Yonezawa and B. C. Smith, editors, *Reflection and Meta-Level Architetures*, pages 36–47. ACM SIGPLAN, 1992.
57. P. P. Pal, J. Loyall, R. E. Schantz, J. Zinky, R. Shapiro, and J. Megquier. Using qdl to specify qos aware distributed (quo) application configuration. In *The 3rd IEEE International Symposium on Object-Oriented Real-time Distributed Computing, (ISORC 2000)*, Mar. 2000.
58. S. Ren. *An Actor-Based Framework for Real-Time Coordination*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.
59. J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings, ACM national convention*, pages 717–740, 1972.
60. H. Rueß. *Formal Meta-Programming in the Calculus of Constructions*. PhD thesis, Universität Ulm, 1995.
61. D. C. Schmidt, D. Levine, and S. Mungee. The design of the Tao real-time object request broker. *Computer Communications*, 21, 1997. Special Issue on Building Quality of Service into Distributed Systems.
62. N. Shankar. *Metamathematics, Machines, and Gödel's Proof*. Cambridge University Press, 1994.
63. B. C. Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, MIT, 1982.

64. C. Smorynski. The incompleteness theorems. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 821–865. North-Holland, 1977.
65. R. M. Smullyan. *Diagonalization and Self-Reference.* Oxford University Press, 1994.
66. D. Sturman. *Modular Specification of Interaction Policies in Distributed Computing.* PhD thesis, University of Illinois at Champaign Urbana, 1996.
67. V. F. Turchin. *Refal-5: programming guide and reference manual.* New England Publishing Co., 1989.
68. N. Venkatasubramanian. *Resource Management in Open Distributed Systems with Applications to Multimedia.* PhD thesis, University of Illinois, Urbana-Champaign, 1998.
69. N. Venkatasubramanian, G. Agha, and C. L. Talcott. Scalable distributed garbage collection for systems of active objects. In *International Workshop on Memory Management, IWMM92, Saint-Malo*, LNCS, 1992.
70. N. Venkatasubramanian, G. Agha, and C. L. Talcott. A formal model for reasoning about adaptive QoS-enabled middleware. In *Formal Methods for Increasing Software Productivity (FME2001)*, 2001.
71. N. Venkatasubramanian and C. L. Talcott. A metaarchitecture for distributed resource management. In *Hawaii International Conference on System Sciences, HICSS-26*, Jan. 1993.
72. N. Venkatasubramanian and C. L. Talcott. Reasoning about meta level activities in open distributed systems. In *Principles of Distributed Computation (PODC '95)*, pages 144–153. ACM, 1995.
73. N. Venkatasubramanian and C. L. Talcott. A semantic framework for modeling and reasoning about reflective middleware, 2001. to appear in Distributed Systems Online.
74. N. Wang, M. Kircher, D. C. Schmidt, and K. Parameswaran. Applying reflective middleware techniques to optimize a QoS-enabled CORBA component model implementation. In *COMPSAC 2000 Conference, Taipei, Taiwan*, 2000.
75. R. W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.
76. A. Yonezawa. *ABCL: An Object-Oriented Concurrent System.* MIT Press, Cambridge Mass., 1990.
77. A. Yonezawa, editor. *Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Reflection 2001.* Lecture Notes in Computer Science. Springer-Verlag, 2001.
78. A. Yonezawa and B. C. Smith, editors. *Reflection and Meta-Level Architecture.* ACM SIGPLAN, 1992.
79. J. Zinky, D. Bakken, and R. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, Apr. 1997.