IEEE Distributed Systems Online
http://computer.org/dsonline
Vol. 2, No. 6, 2001

Reflective Middleware

A Semantic Framework for Modeling and Reasoning about Reflective Middleware

Nalini Venkatasubramanian

University of California, Irvine

Carolyn L. Talcott

Stanford University

Open distributed systems evolve dynamically, and their components interact with environments that are not under their control. A reflective model of distributed computation supports separation of concerns (for example, functionality and different QoS properties) and dynamic adaptation to changing environments or requirements. In such an ODS, a wide range of services and activities must execute concurrently and share resources. To avoid resource conflicts, deadlocks, inconsistencies, and incorrect execution semantics, the underlying resource management system—middleware—must ensure that concurrent system activities compose in a correct manner. Designers and programmers must consider interactions within and across reflective levels, clearly spell out the semantics of shared distributed resources, and develop new notions of overall system correctness that account for a dynamic, distributed, and reflective setting.

To better understand the semantic issues involved in reflective distributed systems, we developed the TLAM,[1–3] a *two-level actor model* based on the actor model of object-based distributed computation.[4–6] *Actors* is a model of distributed reactive objects and has a built-in notion of encapsulation and interaction, making it well suited to represent evolution and coordination among interacting components in distributed applications. Traditional passive objects encapsulate the execution state and a set of procedures that manipulate it; an actor encapsulates a thread of control as well. Each actor potentially executes in parallel with other actors and interacts only by sending and receiving messages.

As the name suggests, in the TLAM, a system is composed of two kinds of actors—*base-level* and *meta-level*—distributed over a network of processing nodes. Base-level actors carry out application-level computation, while meta-level actors are part of the runtime system (middleware) that manages system resources and controls the base-level's runtime semantics. The TLAM abstracts from the choice of a specific programming language or system architecture, providing a framework for reasoning about heterogeneous systems. It supports dynamic customizability and separation of concerns in designing and reasoning about ODS components. In addition, it uses reification (base object state as data at the meta-object level) and reflection (meta-objects modify the base-object state) with support for implicit invocation of meta objects in response to changes of base-level state. This provides for debugging, monitoring, and other services.

TLAM approach

We have used the TLAM framework in several case studies—distributed garbage collection,[7,8] composition of migration and reachability services,[1,3] and QoS-based resource management for multimedia

servers.[9] Our general approach to modeling middleware components is to develop a family of specifications from different viewpoints and at different levels of abstraction. From a high-level viewpoint, we specify the end-to-end service a system provides in response to a request. Expressing systemwide properties in terms of the underlying network's abstract properties can refine this. From a low-level viewpoint, we specify constraints on a group of actors' behavior and distribution. Specifying protocols and algorithms for individual actors' actions can further refine this local behavior viewpoint.

The two viewpoints are related by the notion of a group of meta-actors providing a service in a system that satisfied suitable initialization and noninterference conditions. Staging and refining specifications provides a form of modularity, scalability, and reusability. It reduces the task of implementation to that of implementing individual abstract behaviors. Behavior-level specifications can guide or check implementations or even serve as executable prototypes.

The TLAM framework

The CompOSE|Q (*C*omposable *O*pen *S*oftware *E*nvironment with *Q*oS) framework, currently being developed at the University of California, Irvine,[9,10] implements the basic TLAM meta architecture with additional features to support a reflective communication layer for actor interaction. The work addresses implementation issues and performance overheads associated with realizing the TLAM methodology in a general-purpose middleware framework. (See the "CompOSE|Q Framework" sidebar at the end of this article for more information.)

In the TLAM, meta-actors communicate with each other through message passing, as do base actors, and they examine and modify the state of the base actors located on the same node. Base-level actors and messages have associated runtime annotations (finite sets of tagged values) that meta-level actors can set and read but that are invisible to base-level computations. Actions that change the base-level state are called *events*. In the TLAM, we can change the base-level state by applying a base-level step rule (which involves only base-level entities) or by applying a meta-level step rule (which could involve both base- and meta-level entities). In addition to changing the actors' local state, applying a step rule can create new actors or send new messages. The TLAM event-handling mechanism lets meta-actors react to base-level events.

A TLAM model

A TLAM model is a structure of the form TLAM = <*Net*, *TLAS*, *loc*>, where *Net* is the underlying network, with processor nodes and communication links, and *TLAS* is a two-level actor system with actors distributed over the network by the function *loc*. The TLAM function *loc* maps an actor name to the node on which the actor resides. This corresponds to giving each node a pool of actor names to use when creating new actors, rather than having a global name generator.

The TLAS part of a TLAM model is given by specifying a set *Act* of actor names, a set *Ad* of actor state descriptions,a set *Val* of values that can be communicated in messages and placed in annotations and by specifying step and event-handling rules. Other entities are derived from the three given sets: actors have a name and a state description; messages (*Msg*) have a target actor (an actor name) and content (the value being transmitted); and annotations are finite maps from tags (special values) to values. We let a (and other variables introduced later) range over *Act*, and we let A range over $P_{\omega}(Act)$ (finite sets of actor names)—$a : ad$ denotes an actor a and state ad. We let m range over *Msg*, and we write $a \triangleleft v$ for a message with target a and contents v. The given and derived sets are each partitioned into a base and meta level; $X\lceil\mathtt{b}$ denotes the base-level partition of the set X, and $X\,\mathtt{m}$ denotes its meta-level partition. Base-level actors and messages also have annotations. We write $(a : ad[\alpha])$ for an actor with annotation α—for example, $(a : ad[\mathtt{count = 2}])$ is an actor with a single annotation with tag `count` and value 2.

A step rule describes a possible action for an actor in a given state. It specifies the message consumed (if any) by the stepping actor, the change of the stepping actors state, and any messages to send or actors to create. In addition, meta-level step rules can specify changes in the state of base-level actors, base-level messages to send, and base-level actors to create.

A *meta-actor step rule* has the general form

$$(\dagger)\, a : ad \cdot \mu \xrightarrow[C_b', upd]{C_b} a : ad' \cdot C_{\text{new}} \quad \text{if } cond\ ,$$

where $a : ad$ is the stepping meta-actor with name a in state ad, and μ is either empty or a message $a \lhd v$ to a with content v. The new state of the stepping actor is ad', and C_{new} specifies the newly created meta-level actors and messages. C_b describes the state of the base-level actors that the stepping meta-actor observed; C_b'describes the modification to those actors along with any new base-level actors or messages to create; and *upd* specifies modifications to base-level annotations. *cond* is a predicate that might depend on *a*, *v*, *ad*, and C_b. *Base-level step rules* have the same form as meta-level step rules in which there is no observation or modification of the other base-level state and thus C_b, C_b', and *upd* are omitted. Mathematically, a rule is just a relation on the sorts of entities that appear as components. Often we present these relations using schemata, but the TLAM framework does not specify a particular syntax for rule components such as *ad*, C_b, or *cond*.

A step that delivers or sends base-level messages, changes the base-level state, or creates new base-level actors is represented at the meta-level using events. Formally, an event is a structure containing the observed base-level state, message delivered (if any), and base-level modification. *An event-handling rule* specifies a meta-level actor's response to an event. An event rule is similar to a meta-level step rule with no message delivered and an added event predicate that determines the events to which the rule applies. Furthermore, the only base-level modifications that can be specified are annotation modifications. Applying an event-handling rule does not generate an event.

A meta-actor event-handling rule has the general form

$$\begin{pmatrix}\dagger\\ \dagger\end{pmatrix} a : ad \cdot \mu \xrightarrow[upd]{event} a : ad' \cdot C_{new} \quad \text{if } cond\ ,$$

where ad', C_{new}, *upd*, and *cond* are the same as for step rules, and *event* is the event predicate. We often use patterns to describe event predicates. For example, if *v* is a metavariable ranging over values, and `a` is a particular base-level actor name, then `deliver`(`a` $\lhd$ *v*) describes the predicate that is true for events in which a message is delivered to the actor named `a`.

TLAM semantics: Configurations, transitions, and computations

The semantics of a TLAM model is given by a labeled transition relation on system configurations. A TLAM configuration, *C*, has a set of base-level actors, a set of meta-level actors, and a set of undelivered messages—some are traveling along communication links and others are held in node buffers. Formally, three functions $C = (ac, nq, lc)$, represent a configuration, where *ac* maps the names of actors in the configuration to their state, *nq* maps each node to the sequence of messages in the message buffer of that node, and *lc* maps each link to the sequence of messages in transit in that link. We will represent configurations by listing the actor name–state pairs and message buffers of interest. For example, if $C = (ac, nq, lc)$, then $(a_0 : ad_0) \cdot (a_1 : ad_1) \cdot [\nu : Q] \cdot [\gamma : Q'] \cdot C$ represents a configuration $C' = (ac', nq', lc')$ where ac' is *ac* extended by mapping a_0 to ad_0 and a_1 to ad_1 (we assume that a_0 and a_1 are not in the domain of *ac*); nq' is *nq* extended by mapping node ν to the message sequence Q; and lc' is *lc* extended by mapping link γ to the message sequence Q'. We recall that *loc*(*a*) gives the node on which the actor *a* is located.

We use the following functions to extract information from a configuration *C*:

- *getState*(*C*, *a*) is the state of actor *a* in *C*.
- *getA*(*C*, *a*, *t*) is the value in *C* of the annotation with tag *t* of actor *a*.
- *setA*(*C*, *a*, *t*, *v*) sets the value of the annotation with tag *t* of actor *a*, returning the updated configuration.

Thus, *getA* (*setA* (*C*, *a*, *t*, *v*), *a*, *t*) = *v*.

There are two kinds of transitions on configurations: communication and execution. Communication transitions move undelivered messages from node buffers to links and from links to node buffers and are the same in every TLAM. An execution transition consists of a computation step taken by a base- or meta-level actor, by applying an enabled *step rule*, followed by applying all enabled *event handling rules* (in some order). All actors observed, modified, or created in an execution transition reside on the stepping actor's node.

A rule of the form (†)—that is, a step rule—is enabled in a configuration *C* if

- the meta-actor *a* is present with state *ad*,
- if the message is non-empty, then such a message is available in the message buffer of the node where the actor is located, and
- the condition holds.

A rule of the form (‡)—that is, an event-handling rule—is enabled in a configuration *C* following an application of a step rule if

- the meta-actor *a* is present with state *ad*, on the stepping (base or meta-) actor's node,
- the event predicate *event* holds of the step event, and
- the condition *cond* holds.

When an enabled step or event-handling rule is applied to a configuration, the actor state is changed from *ad* to *ad*′, the message delivered, if any, is removed and the configuration is extended by adding on *loc*(*a*) (the node where *a* is located), the actors, and messages specified by C_{new}. Such a step or rule further modifies the configuration by making the base-level modifications described by C_{b}, C_{b}' (if present) and *upd*—the remainder of the configuration doesn't change. (The transition rules that the TLAS step and event rules induce are spelled out in detail elsewhere.[3])

A computation path is a possibly infinite sequence of labeled transitions. A computation path π is written as a sequence of transitions of the form

$$\pi = \left[C_i \xrightarrow{l_i} C_{i+1} \middle| i \in \text{Nat} \right].$$

The *i*th transition, $\pi(i)$, is

$$C_i \xrightarrow{l_i} C_{i+1}.$$

The source (π (*i*)) of the *i*th transition is C_i—also called the *i*th stage of π. An execution transition's label specifies the stepping actor and the message delivered, if any. A communication transition's label specifies the node and link involved and the direction. For example, `12n`(γ, *v*) labels a transition that moves a message from link γ to node *v*. A configuration's semantics is the set of fair computation paths starting with that configuration. A computation path is fair if

- any communication transition that becomes enabled at some stage eventually occurs;
- any base- or meta-level step enabled at some point eventually happens or becomes permanently disabled; and
- any message in the communication system is either eventually delivered or, from some stage on, no step is enabled to deliver it.

TLAM systems and properties

A TLAM system is a set of configurations closed under the transition relation. Thus, if configuration *C*

is in system S and $C \rightarrow C'$, then C' is also in S. Such a system's properties are specified as predicates on computation paths. A property can be a simple invariant that must hold for all of a path's configurations, a requirement that a configuration satisfying some condition eventually arises, or a requirement involving the transitions themselves. Properties are checked using the properties of the building blocks for configurations—message contents and actor state descriptions—and using properties of the TLAM reaction rules that determine how actors behave in the system.

Preserving base-level computation is a simple example of a property satisfied by meta-level systems that only manipulate base-level annotations—for example, for monitoring. Erasing all meta-level actors and messages and all base-level annotations restricts a configuration to base-level $C\lceil \mathfrak{b}$. Restricting a computation to base-level $\pi\lceil \mathfrak{b}$ is the result of restricting each configuration to base-level and deleting meta-level transitions. Because meta-level actors can change the base-level state, the resulting sequence of transitions might not be a computation.

Definition 1 (Preserving Base-Level Computation): We say that in a system S, base-level computation is preserved if the following holds:

- if π is a fair computation of C, then $\pi\lceil \mathfrak{b}$ is a fair computation of $C\lceil \mathfrak{b}$, and
- if $\pi_{\mathfrak{b}}$ is a fair computation of $C\lceil \mathfrak{b}$, then there is a fair computation π of C such that $\pi_{\mathfrak{b}} = \pi\lceil \mathfrak{b}$.

Preserving base-level computation is a strong requirement. We can define useful notions of preserving base-level behavior by observing only certain aspects of the base-level computation—for example, observing messages exchanged between certain groups of base-level actors and hiding others.

A logger example

In this section, we work out a small example in some detail to illustrate how the TLAM is used to model middleware services as reflective meta-level services. We also illustrate our methodology for specification and proof. The example we use is an incremental logging service that can be installed to log messages that a group of base-level actors receive in a distributed system. The service can also report the logged messages incrementally on request. Because we represent a log as a finite set of messages, we could use such a logging service for accounting purposes. In addition, by annotating the logged messages with sequence numbers or timestamps, we could use the service for fault tolerance.[12]

We begin with a specification of the logging service from an end-to-end point of view. We then define a logging behavior (Definition 7) and the initial and noninterference conditions (Definitions 8 and 9) needed for a system with logging behavior to provide a logging service. Theorem 1 states that logging behavior preserves base-level computation, and Theorem 2 states that under the given initial and noninterference conditions a system with logging behavior does indeed provide logging service. We sketch the proof of this theorem to illustrate how features of the actor model effect reasoning about possible computations—in particular, the use of the fairness assumption—and how to account for information contained in messages in transit as well as information contained in the state of individual actors.

Incremental logs are requested from the logging service using requests of the form *GetLog*(*c*) and replies of the form *LogReply*(*c*, *log*), where *c* is the address to which the incremental log should be sent, and *log* is the increment reported. There should be exactly one reply to each request. Furthermore, each message delivered to a logged actor is reported at most once, and if requests for logs continue to arrive, then each delivered message is eventually reported. We also require the logging service to preserve base-level behavior.

Definition 2 (Incremental Logging Service): A system S provides an *incremental logging service* with respect to request function $GetLog : Act\lceil \mathfrak{m} \rightarrow Msg\lceil \mathfrak{m}$, reply function $LogReply : Act\lceil \mathfrak{m} \times \mathrm{P}_{\omega}(Msg\lceil \mathfrak{b}) \rightarrow Msg\lceil \mathfrak{m}$, and base-level loggable actors $A_{\mathfrak{b}}$, if for each computation

$$\pi = \left[C_i \xrightarrow{l_i} C_{i+1} \middle| i \in \text{Nat} \right]$$

of *S*, the following hold:

1. For each request *GetLog*(*c*) that appears in the undelivered messages of a given C_i, there are uniquely associated j, j' with $i \le j' \le j$ such that the transition $\pi(j')$ delivers that request, and the transition $\pi(j)$ sends a reply *LogReply*(*c, log*).
2. The only messages of the form *LogReply*(*c, log*) sent in π are those associated with a request of the form *GetLog*(*c*) as in statement 1.
3. If a reply *LogReply*(*c*, *log*) is sent during $\pi(j)$, then *log* contains only messages addressed to actors in A_b; *log* is a subset of the messages delivered in π up to stage *j*; and if *LogReply*(*c′*, *log′*) is a distinct log reply sent in $\pi(k)$ for some *k*, then $log \cap log' = \varnothing$.
4. If for each *i* there is some $j \ge i$ such that a new log request appears at stage *j* in π, then every message delivered to an actor in A_b will occur in some log reply.

Note that statements 1 and 2 say that there is a bijection between log requests and log replies (preserving the requestor name).

To specify logging behavior, we specify meta-actor behaviors and configure a service based on these behaviors. We use three kinds of meta-actors:

- a *logger* on each node, which records for the loggable actors on its node,
- a *reporter* on each node, which reports for the loggable actors on its node, and
- a *log server* (one for the system), which receives log requests, collects reports from each node in the system, and sends the log reply.

The logger, reporter, and log server behaviors are each specified by defining a set of states and rules governing their response to messages and events. We define the states by giving constructors, taking the states to be (freely) generated by applying the constructors to appropriate arguments. We begin by defining the data types for the contents of messages used by the meta-actors providing the logger service.

Definition 3 (Logger Messages): There are request-and-reply constructors for communicating between the log server and clients:

- `GetLog` @ _: $Act\lceil \mathtt{m} \rightarrow Val\lceil \mathtt{m}$
- `getLogReply` : $\mathrm{P}_{\omega}(Msg\lceil \mathtt{b}) \rightarrow Val\lceil \mathtt{m}$

Thus, a log request message from a client *c* to a server *ls* has the form *ls* ◁ `getLog`@ *c*.

There are also request-and-reply constructors for communication between the log server and the reporters on each node:

- `sendLog` : $Act\lceil \mathtt{m} \rightarrow Val\lceil \mathtt{m}$, and
- `report(_)` @ _: $\mathrm{P}_{\omega}(Msg\lceil \mathtt{b}) \times Act\lceil \mathtt{m} \rightarrow Val\lceil \mathtt{m}$.

The meta-level actor name in a `report` message is expected to be that of the sender. It is included so a log server managing several reporters will know the reporter from which each reply comes.

Definition 4 (Logger Behavior): There is one constructor for logger states:

Logger: $\mathrm{P}_{\omega}(Act\lceil \mathtt{b}) \rightarrow Ad\lceil \mathtt{m}$.

A logger meta-actor for loggable actors A_b, with name a_l has the form

$(a_l : Logger(A_b))$.

The only logger rule is for handling events that deliver a message to one of its (the logger's) loggable actors:

$$\left(a_l : Logger(A_b)\right) \xrightarrow[setA(a,Log,getA(a,Log)\cup\{a\lhd v\})]{deliver(a\lhd v)} \left(a_l : Logger(A_b)\right) \quad \text{if a} \in A_b$$

This rule says that if $a \lhd v$ is delivered to some $a \in A_b$, then the `Log` annotation of a is updated by adding $a \lhd v$ to its value ($setA(a...)$). The state of a_l doesn't change.

Definition 5 (Reporter Behavior): There is one constructor for reporter states:

$Reporter : \mathrm{P}_\omega(Act \lceil \mathtt{b}) \to Ad \lceil \mathtt{m}$.

A reporter meta-actor for loggable actors A_b with identifier a_r has the form

$(a_r : Reporter(A_b))$.

The only reporter rule is a step rule for handling requests (from the log server) to report its current incremental log:

$$\left(a_r : Reporter(A_b)\right) \cdot a_r \lhd \mathtt{sendLog}(c) \xrightarrow[setA(a,\mathtt{Log},\varnothing)\mid a\in A_b]{} \left(a_r : Reporter(A_b)\right) \cdot c \lhd \mathtt{report}(\log)@a_r$$

where $log = \bigcup_{a\in A_b} getA(a,\mathtt{Log})$

This rule says that when a `sendLog` message is delivered to a_r, then a_r will reply with a `report` message containing the union of the `Log` attributes for actors in A_b, along with its name as the reply sender. The `Log` attributes of each actor in A_b are reset to $\varnothing$ ($setA(a, \mathrm{Log}, \varnothing) \mid a \in A_b$), and the state of a_r doesn't change.

Example logger computation

Before defining the log server behavior, we look at a fragment of a simple computation. This illustrates both the structure of configurations and how rules describe transitions between configurations. C_0 is the computation's initial configuration. It contains, on node ν, a logger a_l, a reporter a_r, two loggable base actors a_0 and a_1, a message for a_0, and other unspecified actors and messages possibly on other nodes (denoted by C_x):

$$C_0 = (a_0 : ad_0[\mathtt{Log} = Q]) \cdot (a_1 : ad_1\,[\mathtt{Log} = \varnothing]) \cdot [\nu : a_0 \lhd v_0] \cdot C_z$$

where $C_z = (a_l : Logger(\{a_0, a_1\})) \cdot (a_l : Reporter(\{a_0, a_1\})) \cdot C_x$.

The state of a_0 in C_0 is ad_0 and a_0 has a single annotation with tag `Log` and value Q. To simplify matters, we assume that the logger and reporter are the only meta-actors on node ν. Transition 0 delivers a message containing v_0 to a_0. The new state ad_0' of a_0 and the message sent $a_2 \lhd v_2$ are computed from ad_0 and v_0 using a base-level step not specified here.

(0) $C_0 \xrightarrow{\mathtt{deliver}(a_0\lhd v_0)} C_1$

where $C_1 = (a_0 : ad_0'\,[\mathtt{Log} = Q \cup \{a_0 \lhd v_0\}]) \cdot (a_1 : ad_1\,[\mathtt{Log} = \varnothing]) \cdot [\nu : a_2 \lhd v_2] \cdot C_z$

The event associated with the base-level satisfies the event predicate `deliver` $a \lhd v$ of the logger event-

handling rule for $a \in \{a_0, a_1\}$. Applying this rule, the delivery is logged using the specified annotation update $setA(a_0, getA(a_0, \texttt{Log}) \cup \{a_0 \triangleleft v_0\})$. As a result, the `Log` annotation of a_0 in C_1 is $Q \cup \{a_0 \triangleleft v_0\}$. (Multi) Transition 1 moves the message to a_2 into the network, and it generates a `sendLog` request and moves it through the network to the head of the link γ, whose target node is v:

(1) $C_1 \xrightarrow{\cdots} C_1'[v:][\gamma : a_r \triangleleft \texttt{sendLog}@\log S] \xrightarrow{12n(\gamma, v)} C_2$

where $C_2 = (a_0 : ad_0'\ [\texttt{Log} = Q \cup \{a_0 \triangleleft v_0\}]) \cdot (a_1 : ad_1\ [\texttt{Log} = \emptyset]) \cdot [v : a_0 \triangleleft v_0] \cdot C_z'$

A transition labeled `12n`(γ, v) moves the message from link γ into the message buffer of node v. C_z' represents changes elsewhere in the system due to the sending of the `sendLog` request (and possibly other activity). Transition 2 delivers the `sendLog` request to a_r, and a reply is sent according to the reporter's step rule:

(2) $C_2 \xrightarrow{\text{exe}(a\ )} C_3$

where $C_3 = (a_0 : ad_0''\ [\texttt{Log} = \emptyset]) \cdot (a_1 : ad_1\ [\texttt{Log} = \emptyset]) \cdot [v : logS \triangleleft \texttt{report}(log)\ @\ a_r] \cdot C_z'$
and $log = Q \cup \{a_0 \triangleleft v_0\}$)

The `Log` annotation of a_0 is set to $\emptyset$ using the update part of the reporter step rule, and $log = getA(C_3, a_0, \texttt{Log}) \cup getA(C_3, a_1, \texttt{Log})$ is the union of the `Log` annotation of the loggable actors.

Definition 6 (LogServer Behavior): A log server meta-actor is either idle and ready to accept log requests, or it is processing a request and waiting for replies from some of its reporters. In both cases, it knows the set of reporters that it manages, and, in the waiting case, it remembers the requestor, the log reports accumulated so far, and the set of reporters whose reports are missing. Thus, we define two state constructors for log server meta-actors:

- `LogServerI` : $\mathrm{P}_\omega(Act\lceil \texttt{m} \rightarrow Ad\lceil \texttt{m}$, and
- `LogServerW` : $\mathrm{P}_\omega(Act\lceil \texttt{m}) \times Act\lceil \texttt{m} \times \mathrm{P}_\omega(Msg\lceil \texttt{b}) \times \mathrm{P}_\omega(Act\lceil \texttt{m}) \rightarrow Ad\lceil \texttt{m}$.

(ls : $LogServerI(R)$) is a log server meta-actor ready to handle a request, and R is the set of log reporter actors that it manages. (ls : $LogServerW(R, c, log, missing)$) is a log server meta-actor waiting for reports, R is the set of log reporter actors that it manages, c is the log requestor, log is the union of the log reports received so far, and $missing$ is the set of reporters for which reports have not yet been received.

There are three log server rules:

(ls : $LogServerI(R)$) $\cdot$ $ls \triangleleft$ `getlog` @ $client \rightarrow$
(ls : $LogServerW(R, client, \emptyset, R)$) $\cdot$ $\{r \triangleleft$ `sendReport` @ $ls \mid r \in R\}$

(ls : $LogServerW(R, client, log, R' \cup \{r\})$) $\cdot$ $ls \triangleleft$ `report`(log') @ $r \rightarrow$
(ls : $LogServerW(R, client, log \cup log', R')$) if $r \notin R'$

(ls : $LogServerW(R, client, log, \emptyset)$) $\rightarrow$
(ls : $LogServerI(R)$) $\cdot$ $client \triangleleft$ `getLogReply`$(log)@ls$

The first rule says that when a log server is idle, it may receive a log request. The request is handled by sending `sendLog` messages to each of its managed reporters and moving to a state in which it waits for a reply from each reporter, with the accumulated log initially empty. The second rule says that when a log server is waiting for reports, it can receive a missing report (r is in the missing set). It adds the reported log

(*log′*) to its accumulated log (*log*) and removes the reporter's name from the missing set. $r \notin R'$ ensures that *r* is really removed. The third rule says that a log server waiting with missing set empty can send the accumulated log to the client and become ready for the next request.

Definition 7 (Logging Behavior Specification): A system *S* has logging behavior with respect to a log server meta-actor *logS*, a set of loggable actors A_b, logging meta-actors *LogMA*(*v*), and reporting meta-actors *ReportMA*(*v*) for $v \in Node$, if for *C* in *S* and $v \in Node$:

- The state of *LogMA*(*v*) in *C* (*getState*(*C*, *LogMA*(*v*))) is *Logger*($A_b \lceil v$).
- The state of *ReportMA*(*v*) in *C* (*getState*(*C*, *ReportMA*(*v*))) is *Reporter*($A_b \lceil v$).
- The state of *logS* in *C* (*getState*(*C*, *LogS*(*v*)) is either *LogServerI*(*R*) or *LogServerW*(*R*, *c*, *log*, *R′*), where $R = \{ReportMA(v) \mid v \in Node\}$, $R' \subseteq R$, $c \in Act\lceil$`m, and log` $\in P_\omega(Msg\lceil$`b`) such that messages in log are addressed to actors in A_b.
- The value of the `Log` annotation (*getA*(*C*, *A*, `Log`)) for each actor *a* in A_b is a set of messages addressed to that actor.

where $A_b \lceil v = \{a \in A_b \mid loc(a) = v\}$.

In the following, we let *LMA* denote the set of log service meta-actors,:

LMA = {*logS*} ∪ *LogMA*(*Node*) ∪ *ReportMA*(*Node*).

A nonlogging meta-actor is thus any meta-actor not in *LMA*.

Theorem 1 (Logging Preserves Base-Level Behavior): If system *S* has logging behavior, then configurations in which the only meta-actors are logging meta-actors of *S* preserve base-level behavior.

Proof: This is easy to see, because meta-actors in *LMA* can only affect base-level annotations.

To state the result that "logging behavior provides logging service," we need two additional definitions: the initial conditions for logging (Definition 8) and the noninterference requirements for logging (Definition 9).

Definition 8 (Logging Initial Conditions): A configuration *C* satisfies the *logging initial conditions* if the value of the `Log` annotation is the empty set for each loggable actor ($a \in A_b$), and the only undelivered messages to logging meta-actors are logging requests addressed to the log server *logS*.

A system *S* satisfies the *logging initial conditions* (relative to the logging behavior parameters) if each configuration is reachable by TLAM transitions for the system from a configuration that satisfies the logging initial conditions.

Definition 9 (Logging Noninterference Requirement): A system *S* satisfies the *logging noninterference requirement* if

- nonlogging meta-actors do not set `Log` annotations
- the only messages sent to logging meta-actors by nonlogging meta-actors are log request messages addressed to the log server *LogS*.

The noninterference conditions are typical of the minimal noninterference conditions for a meta-level service: the annotations a service uses are controlled by that service, and there is no spoofing of communications internal to the meta-actors providing the service.

Now we can state the main result about logging behavior.

Theorem 2 (Logging Behavior Provides Logging Service): If system *S* has logging behavior and satisfies the initial logging conditions and the logging noninterference requirements, then *S* provides

logging service with request

GetLog(*client*) = *LogS* ◁ `getLog`@*client*

and reply

LogReply(*client*,log) = *client* ◁ `getLogReply` (*log*)@*LogS*}.

Proof: Theorem 2 follows from lemmas 1, 2, and 3 (given below). Lemma 1 establishes an invariant on configurations, *LogOk*(*C*), that expresses the key properties of the overall state of a system satisfying the logging initial conditions by accounting for messages in transit as well as local actor states. Lemma 2 establishes responsiveness to requests by showing that the reporting process terminates, leaving the log server ready for another request. Lemma 3 establishes the service's accuracy.

Logger system internal messages are the messages used for communication between logger meta-actors—that is, messages of the form: *ReportMA*(*v*) ◁ *sendLog* @ *logS* or *logS* ◁ *report*(*log*)@ *ReportMA*(*v*).

Definition 10 (LogOk): The invariant *LogOk*(*C*) holds only if the following two conditions hold: First, if the log server is idle, with state *LogServerI*(*R*) in *C*, then there are no undelivered internal logger system messages. Second, if the log server is waiting for reports, with state *LogServerW*(*R*, *c*, *log*, *missing*) in *C*, then there are no undelivered messages to or from the reporters in (*R* – *missing*), and for each reporter *ReporterMA*(*v*) in *missing* exactly one of the following holds:

- there is a single undelivered message of the form *ReporterMA*(*v*) ◁ `sendLog` @ *LogS*, or
- there is a single undelivered message of the form *LogS* ◁ `report` (*log*) @ *ReporterMA*(*v*).

Lemma 1 (LogOK invariance): If *S* satisfies the hypotheses of Theorem 2, then for any configuration *C* of *S*, *LogOk*(*C*) holds.

Proof: By the logging initial condition requirement, every configuration in *S* is reachable in a finite number of transitions—thus, we reason by induction on the number of transitions. If *C* satisfies the logger initial conditions, then we are in the first case of the *LogOk* definition (Definition 10), and initiality guarantees the absence of internal logger messages. Now suppose that *LogOk*(*C*) holds and

$$\tau = C \xrightarrow{l} C'$$

We need only consider execution transitions with a stepping actor in *LMA*, because, using the noninterference requirement, other transitions do not effect the *LogOk* property. If τ delivers a log request to *LogS*, then it must be the case that *LogS* is idle in *C* and waiting in *C′*. Thus, the only internal logger messages in *C′* are the `sendReport` messages sent in the transition, one to each reporter, so *LogOk*(*C′*) holds. If τ delivers a `sendReport` message, then this message is removed, and a `report` message from that reported is added to the undelivered messages. Because no logger meta-actor state changes, *LogOk* is preserved. If τ delivers a report, then the reporter name is removed from *missing* and there are no remaining messages to or from that reporter. Again, *LogOk* is preserved. Finally, if τ ends a log reply to a client, this must be in a situation where there are no undelivered internal logger messages.

Lemma 2 (Termination): If *S* satisfies the hypotheses of Theorem 2, *C* in *S*, and π is a fair path for *C*, then if $\pi(i)$ delivers a log request *logS* ◁ `getLog` @ *c*, then there is some $j > i$ such that in $\pi(j)$, a reply *c* ◁ `getLogReply`(*log*) @ *logS* is sent.

Proof: This follows from the fairness of the message delivery system and the fact that each reporter sends a reply in the same transition that a `sendLog` request is delivered.

To state the accuracy lemma, we define two functions on computation paths to keep track of the accumulated deliveries and log reports. $\Delta(\pi,i)$ is the set of messages delivered to an actor in A_b in transitions $\pi(j)$ for $j < i$. $L(\pi,i)$ is the union of the set of messages sent to clients in log request replies in transitions $\pi(j)$ for $j < i$. Logging accuracy is a consequence of the relation between Δ and L, established in the following lemma.

Lemma 3 (Logging accuracy): If S satisfies the hypotheses of Theorem 2, C in S, and π is a fair path for C, then for any $i \in$ `Nat`, letting C_i be the source configuration of $\pi(i)$, we have that $L(\pi, i) \subseteq \Delta(\pi, i)$ and for any $m \in (\pi, i)$, one of the following cases holds:

1. m is in the `Log` annotation of its target actor,
2. there is an undelivered internal logger report $LogS \lhd$ `report`(log) @ `ReporterMA`(ν) such that $m \in log$ (and the target of m is located on ν),
3. the state of $logS$ in C_i is $logServeW(R, c, log, R')$ with $m \in log$, a or
4. $m \in L(\pi, i)$.

Furthermore, if cases 2 or 3 hold, or if a log request arrives after stage i, then at some stage $j > i$, case 4 will hold.

Proof: The proof is by induction on i. The case $i = 0$ is trivial since$L$ and Δ are empty. Assume that the conditions hold for i. If $\pi(i)$ delivers m to $a \in A_b$, then $\Delta(\pi, i + 1) = \Delta(\pi, i) \cup \{m\}$ and condition 1 holds for m. `sendReport` delivery transitions move messages from condition 1 to condition 2, `report` delivery transitions move messages from condition 2 to condition 3, and transitions that send a reply to a client move messages from condition 3 to condition 4. Other transitions do not effect L or Δ.

Currently, TLAM reflection provides support for implicit invocation of meta-objects in response to changes of base-level state. We want to extend the model to support signals from the base to the meta-level to allow for the explicit invocation of meta-level services by base objects.

We also want to extend the TLAM to provide for specification of a wider range of meta-level services, including scheduling, routing, name services, failure semantics, and security. For example, by modeling the scheduler explicitly and providing access to it, we can introduce the notion of priorities, preemption, and real time to the two-level system. Using generalized state-capture facilities within the TLAM framework, we are developing support for fault-tolerant systems—for example, a checkpointing service for capturing causal orders of executions in the system that can help monitor and debug distributed computations. We are also exploring using a directory core service to develop policies for resource discovery, group-based communication, access control, and security.

References

1. N. Venkatasubramanian and C.L. Talcott, "Reasoning about Meta Level Activities in Open Distributed Systems," *14th ACM Symp. Principles of Distributed Computing*, ACM Press, New York, 1995, pp. 144–152.
2. N. Venkatasubramanian, *An Adaptive Resource Management Architecture for Global Distributed Computing*, PhD thesis, Department of Computer Science**,** University of Illinois, Urbana-Champaign, 1998.
3. N. Venkatasubramanian and C.L. Talcott, *Integration of Resource Management Activities in Distributed Systems*, tech. report draft, Computer Science Dept., Stanford Univ., 2001.
4. C. Hewitt, P. Bishop, and R. Steiger, "A Universal Modulal Actor Formalism for Artificial Intelligence," *Proc. 1973 Int'l Joint Conf. Artificial Intelligence*, 1973, pp. 235–245.
5. H.G. Baker and C. Hewitt, "Laws for Communicating Parallel Processes," *IFIP Congress*, 1977, pp. 987–992.
6. G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, Mass., 1986.
7. N. Venkatasubramanian, G. Agha, and C.L. Talcott, "Scalable Distributed Garbage Collection for Systems of Active Objects," *Int'l Workshop Memory Management* (IWMM92), *Lecture Notes in Computer Science*, Springer Verlag, 1992.
8. N. Venkatasubramanian, *Hierarchical Garbage Collection in Scalable Distriubted Systems*, master's thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, 1992.
9. N. Venkatasubramanian and C.L. Talcott, "A Formal Model for Reasoning about Adaptive Qos-Enabled Middleware," *Formal Methods Europe*, 2001.
10. N. Venkatasubramanian, "Composeq: A Qos-Enabled Customizable Middleware Framework for Distributed Computing," *Proc. Middleware Workshop, Int'l Conf. Distributed Computing Systems* (ICDCS99), 1999.
11. N. Venkatasubramanian et al., "Design and Implementation of a Composable Reflective Middleware Framework," *Proc. IEEE Int'l Conf. Distributed Computing Systems* (ICDCS2001), 2001.
12. L. Alvisi and K. Marzullo, "Message Logging: Pessimistic, Optimistic, Causal and Optimal," *IEEE Trans. Software Eng.*, vol. 24, no. 2, 1998.

Nalini Venkatasubramanian is an assistant professor in the Department of Information and Computer Science at the University of California, Irvine. Her research interests include distributed and parallel systems, middleware, real-time multimedia systems, mobile environments, and formal reasoning of distributed systems. She is specifically interested in developing safe and flexible middleware technology for highly dynamic environments. She has an MS and PhD in Computer Science from the University of Illinois, Urbana-Champaign, and she is a member of the IEEE and ACM. Contact her at nalini@ics.uci.edu.

Carolyn Talcott is a Senior Research Associate in the Computer Science Department at Stanford University. She is interested in formal reasoning and in semantic foundations for open distributed systems. She has a PhD in chemistry from UC Berkeley and a PhD in computer science from Stanford University. She is Co-Editor-in-Chief of *Higher-Order and Symbolic Computation*. She is a member of ACM and IFIP WG6.1 and a co-founder ETraffic Inc. Contact her at clt@cs.stanford.edu.

The CompOSE|Q framework

The CompOSE|Q (*C*omposable *O*pen *S*oftware *E*nvironment with *Q*oS) framework (see Figure A), currently being developed at the University of California, Irvine(add refs) implements the basic TLAM meta-architecture with additional features to support a reflective communication layer for customizable actor interaction. The work addresses implementation issues and performance overheads associated with the realization of the TLAM methodology in a general purpose middleware framework. To ensure noninterference and to manage the complexity of reasoning about multiple middleware components and services, we identify three core services within CompOSE|Q that meta-actors provide where nontrivial, base meta-interactions occur: remote creation, distributed snapshots,[1,2] and directory services. Using the core services, we define other meta-level services and specify properties such as functionality, QoS, and noninterference requirements in terms of purely meta-level interactions, which are easier to express and understand but are still nontrivial.

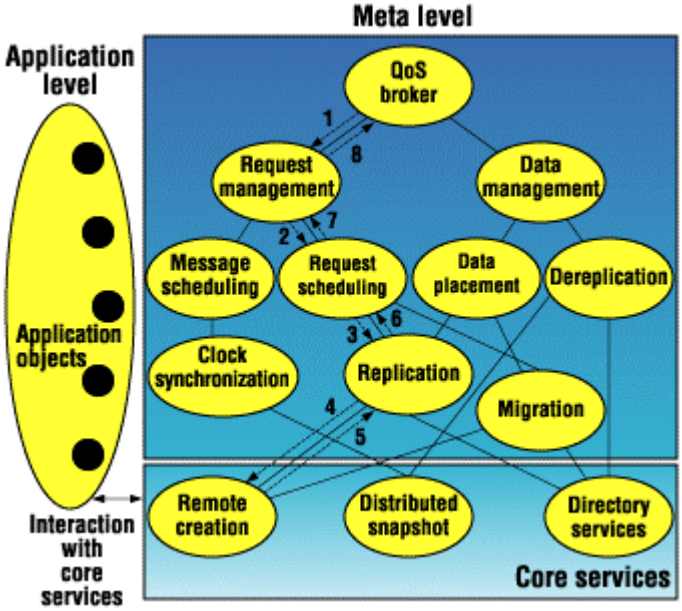

Figure A. Architecture of the CompOSE|Q system. The dotted lines indicate the flow of an incoming request through the middleware modules.

Within CompOSE|Q we have developed modules that implement resource management services such as remote creation, migration, and reachability snapshots from formally verifiable specifications developed using the TLAM. The services implement necessary invariants and constraints to safely compose meta-level services. The design, implementation and performance of a node-based runtime layer and a distribution infrastructure that implements the necessary TLAM abstractions (meta-actors, base-actors, core services) along with suitable optimizations to improve the performance of TLAM-based resource management services are described elsewhere.

To ensure safe and cost-effective QoS in distributed environments, composability (concurrent execution) of resource management services such as admission control, resource reservation, and scheduling is essential. For instance, system-level protocols should not cause arbitrary delays in the

presence of timing-based QoS constraints. The CompOSE | Q framework implements QoS-based resource management services for distributed multimedia systems as meta-level services in the reflective architecture. Here, we use services developed earlier within the TLAM framework (such as migration and snapshot primitives) to develop a QoS brokerage architecture and to reason about the how services interact within the architecture.[3,4]

We apply the QoS broker system (see Figure A) to implement the scheduling and routing of multimedia requests as well as placement of multimedia data using a collection of multimedia resource management meta-actors that provide these services and show how to reason about their interaction (add reference). The TLAM framework also provides a semantic basis for developing tools that support modular design and analysis of middleware services with increased assurance. We are working on realizing the TLAM framework using the Maude tool,[5] which is based on rewriting logic.[6] Thus, a TLAM in Maude constitutes an executable specification that can be used for prototyping, symbolic simulation, and simple analyses (using reflection in Maude).

References

1. K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of a Distributed System, *ACM Trans. Computing Systems*, vol. 3, no. 1., 1985, pp. 63–75.
2. K.M. Chandy and J. Misra, *Parallel Program Design*, Addison-Wesley, Reading, Mass., 1988.
3. N. Venkatasubramanian and C.L. Talcott, "A Formal Model for Reasoning about Adaptive Qos-Enabled Middleware," *Formal Methods Europe*, 2001.
4. N. Venkatasubramanian, "Composeq: A Qos-Enabled Customizable Middleware Framework for Distributed Computing," *Proc. Middleware Workshop, Int'l Conf. Distributed Computing Systems* (ICDCS99), 1999.
5. J. Meseguer, "A Logical Theory of Concurrent Objects and its Realization in the Maude Language," *Research Directions in Object-Based Concurrency*, G. Agha, P. Wegner, and A. Yonezawa, eds., MIT Press, Cambridge, Mass., 1993.
6. J. Meseguer, "Conditional Rewriting Logic as a Unified Model of Concurrency," *Theoretical Computer Science*, vol. 96, no. 1, 1992, pp. 73–155.

Related Work

Research on computational reflection was initiated by the work on 3-Lisp (reference) and 3KRS(reference). 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 explores different forms of reflection, including single-actor and group-based reflection.[1] A layered reflection model (the *onion skin model*) is a basis for developing coordination primitives,[2,3] representing dependability protocols as meta-level programs[4,5] and architectural abstractions.[6] Meta-object protocols[7] provide more restricted forms of reflective capability, consisting of interfaces to a language that give users the ability to incrementally modify the language's behavior and implementation, as well as the ability to write programs within the language. This approach was generalized to the Aspect Oriented Programming paradigm[8] to facilitate separation of concerns, composition, and reuse in programming.

In other reflective models for distributed object computation—for example, the MultiModel Reflective Framework (MMRF)[9]— multiple models represent an object, allowing behavior to be described at different levels of abstraction and from different viewpoints. In each model, the behavior of an object is described by a metaspace that consists of meta-objects representing the different models and aspects (such as encapsulation or environment). Each meta-object sees and acts on one base-level object. In the TLAM, each meta-actor can examine and modify the behavior of a group of base-level actors—namely, those located on the same node. While some instances of the TLAM might have the many-to-one organization of the multiple model frameworks, the more flexible relation seems appropriate when considering resource management facilities that manipulate collections of base-level actors.

Examples of operating systems built using reflective distributed object models are Apertos,[10] Legion,[11] and 2K.[12] Adaptability and extensibility is a prime requirement of middleware systems; reflective middleware typically builds on the idea of a meta-object protocol, with a meta-level describing the middleware's internal architecture and reflection used to inspect and modify internal components. DynamicTao[13] is a reflective CORBA ORB[14] built as an extension of the Tao real-time CORBA ORB.[15] DynamicTao supports on-the-fly reconfiguration while maintaining consistency by reifying both internal structure and dependency relations using objects called configurators, which provide interfaces for inspection and modification. Other work on using reflective ORBs to customize resource management behavior, such as scheduling, is reported elsewhere.[16] Other work uses reflective middleware techniques to enhance adaptivity in QoS-enabled component-based applications using the Tao ORB.[17] Costa et al describes a reflective architecture for next-generation middleware based on multiple metamodels,[18,19] and a prototype implementation using Python's reflective capabilities.[20]

One of the main objectives of the TLAM architecture is to develop formal models, reasoning and analysis techniques to address the challenges of performance and integrity management associated with adaptable and extensible middleware. An associated goal is to provide a semantic basis for separation of concerns and compositional development of middleware for large-scale distributed systems. There are two dimensions of composition: that of the base-level with the meta-level and composition of meta-level services over arbitrary base-level systems. Work on superposition[21] also addresses issues of separating concerns and of compositionality by developing generic linguistic constructs based on notions of role and binding. An interesting question for future investigation is whether we can use the notion of superposition to develop more systematic composition mechanisms within the TLAM framework.

The two-level TLAM architecture naturally extends to multiple levels, with each level manipulating the level below while being protected from manipulation by lower levels. A purely reflective architecture provides an unbounded number of meta-levels with a single basic mechanism. Formally verifying interaction semantics between the different layers in the reflective hierarchy can be quite complex. As an initial step, we have started focusing on the special case of two levels. The challenge here is to develop easy-to-use principles for harnessing the power of reflection and to avoid the potential chaos that is possible with its unrestricted use. This article's objective is to illustrate use of the TLAM framework, and the key elements of its approach to modeling middleware services from multiple viewpoints. You can find a full mathematical definition of the TLAM framework elsewhere, along with some detailed case studies.[22]

References

1. A. Yonezawa, *ABCL: An Object-Oriented Concurrent System*, MIT Press, Cambridge, Mass., 1990.
2. S. Frølund, *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization*, MIT Press, Cambridge, Mass., 1996.
3. G. Agha et al., "Abstraction and Modularity Mechanisms for Concurrent Computing," *IEEE Parallel and Distributed Technology: Systems and Applications*, vol. 1, no. 2, May 1993, pp. 3–14.
4. G. Agha et al., "A Linguistic Framework for Dynamic Composition of Dependability Protocols," *Proc. 3rd IFIP Working Conf. Dependable Computing for Critical Applications*, 1992.
5. D. Sturman, *Modular Specification of Interaction Policies in Distributed Computing*, PhD thesis,Department of Computer Science, , Univ. of Illinois, at Urbana-Champaign, 1996. .
6. M. Astley, *Customization and Composition of Distributed Objects: Policy Management in Distributed Software Architectures*, PhD thesis, Department of Computer Science, Univ. of Illinois, Urbana-Champaign, 1999.
7. G. Kiczales, J. des Riviers, and D. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, Cambridge, Mass., 1991.
8. G. Kiczales et al., "Aspect-Oriented Programming," *Proc. ECOOP'97 European Conf. Object-Oriented Programming*, 1997.
9. H. Okamura, Y. Ishikawa, and M. Tokoro, "Al-1/d: A Distributed Programming System with Multi-Model Reflection Framework," *Reflection and Meta-Level Architetures*, A. Yonezawa and B.C. Smith, eds., 1992, pp. 36–47.
10. J.-I. Itoh, Y. Yokote, and R. Lea, "Using Meta-Objects to Support Optimisation in the Apertos Operating System," *USENIX Conf. Object-Oriented Technologies* (COOTS), Usenix, 1995, pp. 147–158.
11. A. Grimshaw et al., "The Legion Vision of a Worldwide Virtual Computer," *Comm. ACM*, vol. 40, no. 1, Jan. 1997.
12. F. Kon et al., "2K: A Reflective, Component-Based Operating System for Rapidly Changing Environments," *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, 1998.
13. F. Kon et al., "Monitoring, Security, and Dynamic Configuration with the DynamicTAO Reflective ORB," *Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms and Open Distributed Processing* (Middleware'2000), *Lecture Notes in Computer Science*, no. 1795, Springer-Verlag, New York, Apr. 2000, pp. 121–143.
14. Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.3 ed., June 1999.
15. D.C. Schmidt, D. Levine, and S. Mungee, "The Design of the Tao Real-Time Object Request Broker," *Computer Comm.*, vol. 21, 1997.
16. A. Singhai, A. Sane, and R. Campbell, "Reflective ORBs: Support for Robust, Time-Critical Distribution," *Proc. ECOOP'97 Workshop on Reflective Real-Time Object-Oriented Programming and Systems*, 1997.
17. N. Wang et al., "Applying Reflective Middleware Techniques to Optimize a QoS-Enabled CORBA Component Model Implementation," *COMPSAC 2000*, 2000.
18. F. Costa, G. Blair, and G. Coulson, "Experiments with Reflective Middleware," *European Workshop on Reflective Object-Oriented Programming and Systems* (ECOOP '98), Springer-Verlag, New York, 1998.
19. G. Blair et al., "An Architecture for Next Generation Middleware," *Middleware '98*, 1998.
20. A. Andersen, *A Note on Reflection in Python 1.5*, tech. report MPG-98- 05, Distributed Multimedia Group, Lancaster Univ., 1998.
21. S. Katz, "A Superimposition Control Construct for Distributed Systems," *ACM TOPLAS*, vol. 15, no. 2, Apr. 1993, pp. 337–356.
22. N. Venkatasubramanian and C. L. Talcott, *Integration of Resource Management Activities in Distributed Systems*, tech. report draft, Computer Science Dept., Stanford Univ., 2001.