

Customization and Composition of Distributed Objects: Middleware Abstractions for Policy Management

Mark Astley and Gul A. Agha
Department of Computer Science
Univ. of Illinois at Urbana-Champaign
1304 W. Springfield, Urbana, IL, 61801, USA
{astley, agha}@cs.uiuc.edu

Abstract

Current middleware solutions such as CORBA and Java's RMI emphasize compositional design by separating functional aspects of a system (*e.g.* objects) from the mechanisms used for interaction (*e.g.* remote procedure call through stubs and skeletons). While this is an effective solution for handling distributed interactions, higher-level requirements such as heterogeneity, availability, and adaptability require policies for resource management as well as interaction. We describe the *Distributed Connection Language* (DCL): an architecture description language based on the Actor model of distributed objects. System components and the policies which govern an architecture are specified as encapsulated groups of actors. Composition operators are used to build connections between components as well as customize their behavior. This customization is realized using a meta-architecture. We describe the syntax and semantics of DCL, and illustrate the language by way of several examples.

1 Introduction

The complexity of modern distributed systems has led to an emphasis on compositional system development. For example, middleware solutions such as the Common Object Request Broker Architecture (CORBA) [9] and Java's Remote Method Invocation (RMI) [14] have incorporated compositional design by separating functional aspects of a system (*e.g.* objects) from the mechanisms used for interconnection (*e.g.* remote procedure call through stubs and skeletons). Separating objects from the policies which govern their interaction simplifies debugging and makes reuse feasible. In particular, the protocols required for interaction need not be hard-coded in objects [11]. This allows objects and protocols to be independently tested and later composed into runnable systems. Moreover, as requirements change, existing architectural elements may be modularly replaced by new elements with appropriate properties.

Approaches such as CORBA and RMI allow for more efficient system development, but still lack tools for verifying that compositions are semantically correct: stubs and skele-

tons may be used to connect objects but do not guarantee that interfaces are invoked properly. To verify correctness and reason about composability, researchers have introduced the notion of *architecture description languages* (ADLs). An ADL specification defines a software architecture in terms of a collection of *components*, which encapsulate computation, and a collection of *connectors*, which describe how components are integrated into the architecture [10].

Current ADLs emphasize modular specification of components and their interaction. While this emphasis has demonstrated advantages for system development, in general, distributed systems entail more complicated behavior. In particular, heterogeneity, failure, and the potential for open (*i.e.* unpredictable) interactions yield evolving systems which require complex management policies. Note that such policies are not isolated to interactions between components. For example, consider a distributed client-server architecture. While connectors define the mechanisms by which clients are matched to the server, the nature of the execution environment may require additional policies for system management:

- **Cluster Management:** The server may execute on a dedicated cluster for high-availability. Thus, we might require a cluster resource policy which manages the allocation of cluster resources among server components.
- **Secure Interactions:** Clients may interact with the server over an insecure network. To provide secrecy, we may require an encryption policy which is enforced over connectors linking clients to the server.

Policies such as cluster management and encryption assert properties associated with the execution environment, and are orthogonal to the functional behavior of the system. While it is possible to embed such policies within components and connectors, doing so sacrifices modularity in the same way that embedding interaction mechanisms within objects sacrifices object modularity. A more desirable solution is to develop a new model of components and connectors which exposes architectural features such as resource usage and locality, and which provides new abstractions for asserting policies over these features.

In this paper, we describe the *Distributed Connection Language* (DCL): an Actor-based architecture description language for specifying distributed systems. While common architectural styles may be specified in DCL, we place particular emphasis on the dynamic behavior associated with distributed systems. DCL abstractions are based on collections of actors. Linguistically, a DCL specification is rule

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGSOFT '98 11/98 Florida, USA
© 1998 ACM 1-58113-108-9/98/0010...\$5.00

based and reactive, specifying changes to an architecture in response to run-time interactions. Architectural policies are defined as meta-level customizations applied to collections of actors. Policy composability is facilitated by a composable meta-architecture.

The remainder of this paper is organized as follows. In the latter part of this section, we describe the Actor model and related work in architecture description languages. In Section 2, we describe *modules* and *protocols*, which serve as the basic building blocks of a distributed architecture. We then describe customization of modules and protocols using architectural *policies* in Section 3. Finally, in Section 4, we summarize our results and discuss topics for future research.

1.1 Actors

We use Actors [1] as a basis for modeling distributed software architectures. Actors provide a general and flexible model of concurrency. As an atomic unit of computation, actors may be used to build typical architectural elements including procedural, functional, and object-oriented components. Moreover, actor interactions may be used to model standard distributed coordination mechanisms such as remote procedure call (RPC), transactions, and other forms of synchronization [2].

Conceptually, an actor encapsulates a state, a thread of control, and a set of procedures which manipulate the state. Actors coordinate by asynchronously sending messages to one another. Each actor has a unique *mail address* and a *mail buffer* to receive messages. Actors compute by serially processing messages queued in their mail buffers. An actor blocks if its mail buffer is empty. While processing a message, there are three basic actions which an actor may perform that affect the computational environment:

- *send* messages asynchronously to other actors;
- *create* actors with specified behaviors; and
- become *ready* to receive the next message.

Communication is point-to-point and is assumed to be weakly fair: executing a *send* eventually causes the message to be buffered in the mail queue of the recipient although messages may arrive in an order different from the one in which they were sent. The *create* primitive creates a new actor with a specified behavior. Initially, only the creating actor knows the name of the new actor. However, actor names are first class entities which may be communicated in messages. Thus, coordination patterns between actors may be dynamic. The *ready* primitive is used to indicate that the signaling actor is ready to process the next message in its mail queue. Upon invoking *ready*, the calling actor either begins processing the next available message, or blocks until a new message arrives.

Note that modern sequential languages are readily extended with the actor primitives (*c.f.* [15]). In many instances, this mechanism may be used to incorporate legacy software into DCL architectures.

1.2 Related Work

Recently, several architecture description languages have been proposed in the literature. Of these languages, Rapide [8] and Wright [3] are useful representatives due to their formal nature and the scope of their specifications. We describe Rapide and Wright here, and refer the reader to [5] for an

interesting survey of architecture description languages in general.

Rapide is an object-oriented language designed for event-based prototyping of distributed software architectures. A Rapide *architecture* contains a set of module specifications called *interfaces* which define a collection of named entry points. The behavior of an architecture is defined by a set of *connection rules* which determine how events are transmitted between interfaces. The set of *formal constraints* for an architecture restricts the patterns of events transmitted by connection rules. Formal constraints specified on an interface provide a “contract” describing the context of an interface in an architecture. This represents the simplest form of connection policy between two modules: that implied by their respective local constraints.

In contrast, the Wright language defines architectural structure in terms of extensions to Communicating Sequential Processes (CSP) [6]. A Wright *architecture* consists of an interlinked set of components and connectors. Wright components are specified by an *interface* and a *computation*. An interface describes a fixed set of *ports* and their behavior. A computation describes component behavior in terms of interactions triggered at ports. A Wright connector defines an interaction pattern between a set of ports in terms of *roles*, which describe the behavior of each participant in the connection, and *glue* which describes how the participants are linked to define an interaction. A role represents the behavior expected by an interface port which assumes the role. The glue of a connector is a complete behavioral specification of how events from one port are translated into events on another port.

A key difference between DCL, and Rapide and Wright is the emphasis on dynamically reconfigurable architectures. Characterizing dynamic behavior is particularly important in the context of distributed systems, where connectivity is often a run-time rather than compile-time property. Thus, DCL specifications are rule-based and define potential architectural structures in response to run-time interactions. At the time of this writing, Wright specifications are strictly static. Rapide, however, does provide support for representing dynamic connectivity properties.

Another important difference is the use of a meta architecture for exposing and customizing the internal resource usage of architectural elements. This is necessary in order to specify policies which customize interactions and resource acquisition of components and connectors. In contrast, Wright and Rapide do not specify component semantics beyond a functional interface. Instead, emphasis is placed on connection mechanisms, which are completely specified in both Wright and Rapide.

Although we believe our approach to be novel in the realm of architecture description languages, our abstractions for meta-architecture and composition are similar to techniques for protocol development in systems such as the x-Kernel [7] and Horus [16]. In particular, our methods for protocol composition are essentially protocol stacks as modeled in these systems. We differ from protocol-based work, however, by extending our approach to include policies for module resource management as well as interactions.

2 Basic Architectural Specification

An architectural unit in DCL represents an abstraction over an encapsulated collection of actors. These collections are encapsulated in the sense that they form a private namespace. In particular, while actor names may be freely ex-

changed within messages, the implementation of DCL prevents the delivery of messages where the sender and receiver do not reside in a common collection. Collections may interact, however, by sharing a common set of members. Specifically, an *admission* modifies the membership of a collection by including an actor from another collection. The admitted actor becomes a member of both collections and may be used to exchange messages between actors in each collection. Actor creation is handled as a special case of admission where a new actor is automatically admitted to the collection of its creator.

The syntax of DCL is used to define the initial members of a collection and the conditions under which admissions are performed. In particular, a basic specification in DCL consists of two types of structures:

- **Module:** A module defines a computational unit within an architecture. Actors within a module define the behavior of the computation. Interactions between modules are handled by exchanging messages through protocol actors, which are provided by protocol connections.
- **Protocol:** A protocol defines an interaction mechanism between modules. Internally, protocols consist of a collection of actors which are assigned to “roles.” Protocol actors are “submitted” to modules when a protocol is used to build a connection.

In ADL terminology, a *module* is a component and a *protocol* is a connector. Traditionally, components and connectors have a limited number of connection points and architectures are static structures fixed at specification time. In contrast, DCL specifications have both static and dynamic aspects. In particular, modules and protocols are rule-based, and DCL architectures are reconfigurable at runtime. Without loss of generality, we restrict ourselves to the dynamic aspects of DCL for this presentation. Note that static architectural configurations may be viewed as an abstraction over the “bootstrapping” phase of a purely dynamic architecture¹.

2.1 Modules

A module encapsulates a collection of actors (called *module actors*) which implement a particular computational behavior. As with actors, each module instance has a unique name which is used to interact with the module. Module names are the only externally visible references in a module. That is, while the actors within a module form a closed namespace, module names are global identifiers which allow external protocols and policies to initiate connections.

Figure 1 gives an abstract syntax for modules. A module specification consists of an id and a body. A module id is used as a type identifier when new modules are instantiated. The body of a module is divided into two sections:

- **Local State:** Local state consists of a fixed set of local variables with simple types (i.e. integer, string, array, etc.), or references to local actors and/or external modules and protocols. An initialization section may be defined to initialize the local state when the module or protocol is created.

¹Static architectures also have the benefit of allowing compile-time type checking. We plan to discuss the static aspects of DCL (including compile-time type properties) in a future publication.

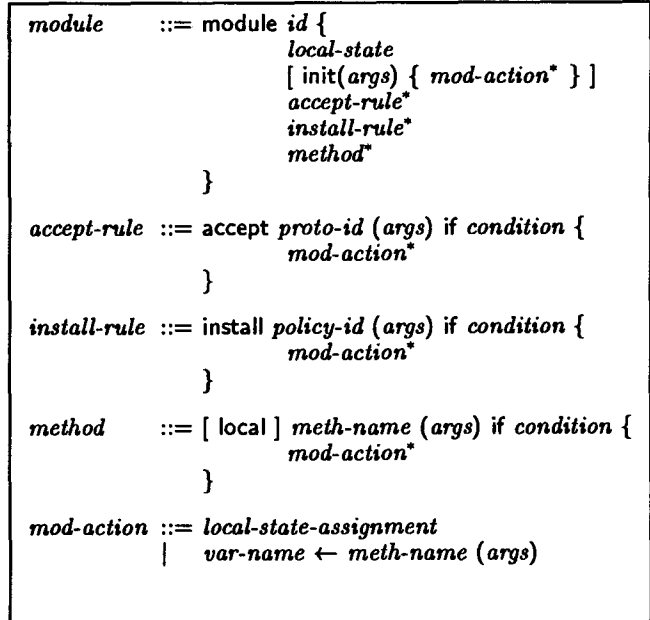


Figure 1: **Module Syntax:** A module defines local state, an initialization section, and a set of request rules. An accept rule is used to process connection requests. An install rule is used to allow the customization of internal actors. Methods are used for coordination and synchronization.

- **Request Rules:** Request rules define the control interface of a module. In general, a rule consists of a type, a caller id, place holders for parameters, a boolean condition, and a rule body. An accept rule is used to accept a connection request from an external protocol. An install rule is used to allow policies to customize the internal actors of a module. A method is a general rule which allows coordination among local module members and external modules or protocols.

Rules are matched in the order of their specification. Only the first matched rule is invoked. If no rule is matched by a request, then the request is ignored. Rule bodies are strictly declarative. Within a rule body, two actions are possible: local state may be assigned, or messages may be sent to other entities. Local state definitions consist of a type and a variable name. Variables are assigned in the usual fashion (i.e. *var-name* := *val*). New actors, modules or protocols may be instantiated and assigned to local variables using the syntax:

```
var-name := new type-name (args)
```

If *type-name* is an actor type, then the actor is created locally. Otherwise, *type-name* refers to a module or protocol, and the new entity is created externally. *Var-name* is set to the name of the new entity after it has been created. The remainder of the syntax is defined as follows:

- accept proto-id (args) if condition { mod-action* }
Defines an accept rule. An accept rule is matched if a request originates from a protocol of type *proto-id*

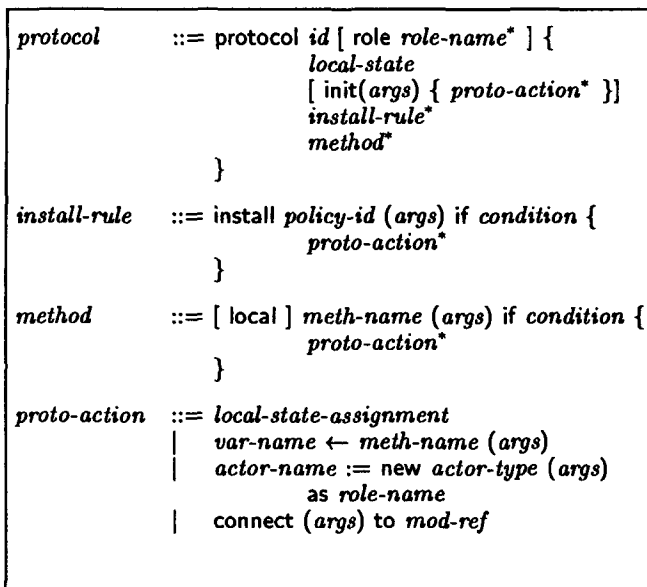


Figure 2: **Protocol Syntax:** A protocol defines local state, an initialization section, and a set of request rules. Actors created within a protocol are assigned to a role at the time of creation. The connect keyword is used to submit a connection request to an external module.

with parameters matching the type signature of *args*, and *condition* is satisfied. *Condition* is a boolean expression evaluated over the local state of the module and *args*. If an accept rule is matched then *args* is bound to the request parameters and the rule body is evaluated. If *args* contains a reference to a protocol actor of the calling protocol, then this actor is automatically admitted to the module's namespace before the body is evaluated (see Section 2.2).

- `install policy-id (args) if condition { mod-action* }`

Defines an install rule. An install rule is matched if a request originates from a policy of type *policy-id* with parameters matching the type signature of *args*, and *condition* is satisfied. *Condition* is a boolean expression evaluated over the local state of the module and *args*. If an install rule is matched then *args* is bound to the request parameters and the rule body is evaluated. A matched rule also results in the installation of a policy actor on each actor in the module. We describe this process in more detail in Section 3.
- `[local] meth-name (args) if condition { mod-action* }`

Defines a method. A method is matched if a request specifies the target *meth-name* with parameters matching the type signature of *args*, and *condition* is satisfied. *Condition* is a boolean expression evaluated over the local state of the module and *args*. We enforce the additional constraint that the request must originate from either a local actor or an external module or protocol. If the keyword *local* is present, then the rule only matches requests sent by internal actors. If a method rule is matched then *args* is bound to the

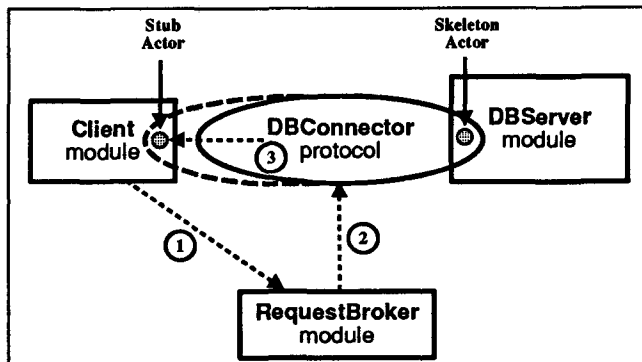


Figure 3: **A CORBA-like Client-Server Architecture:** A database server creates a connection protocol and obtains a skeleton for receiving interactions.

parameters of the message and the rule body is evaluated.

- `var-name ← msg-name (args)`

Creates an interaction. An interaction causes a message to be sent if *var-name* refers to a local actor, or invokes a method if *var-name* refers to a protocol or policy. *Msg-name* identifies the method to invoke on the target and *args* parameterizes the message. Interactions always occur asynchronously (*i.e.* the caller is not blocked).

Accept and install rules modify the namespace of a module in order to form new connections or enforce architectural policies. In the case of an accept rule, one or more *protocol actors* are admitted as endpoints for a connection to another module. In the case of an install rule, several *policy actors* are admitted as meta-level customizations of internal module actors.

2.2 Protocols

A protocol encapsulates a collection of actors (called *protocol actors*) which govern the interactions between a set of modules. A protocol connection is created by admitting one or more protocol actors to the namespace of each connected module. Protocol actors admitted in this fashion become members of both namespaces, and may communicate with actors in either space.

Syntactically, protocols are similar to modules except that a protocol definition must also include a fixed number of named "roles". Roles are meant to indicate the organization of a protocol. For example, a UNIX-like pipe protocol would have a *source* role, where interactions originate, and a *sink* role where interactions are delivered. Moreover, a special syntax is used to ensure that each actor created by a protocol is associated with one of the roles declared in the protocol specification (see Figure 2). The connect action is provided to submit connection requests. Protocol syntax which differs from that of modules is defined as follows:

- `actor-name := new actor-type (args) as role-name`

Instantiates a new actor and assigns its reference to a local state variable. The type of the new actor is

```

module DBServer {
  boolean connected = false;
  module broker;
  protocol connector;
  init(module rBroker) {
    Create internal resources;
    // Save reference to request broker
    broker := rBroker;
    // Create the connection protocol and request a skeleton
    connector := new DBConnector();
    connector ← connectSkeleton(self);
  }
  // Only accept one skeleton
  accept DBConnector(actor skeleton) if !connected{
    Forward skeleton to appropriate local actors;
    // Register server with request broker
    broker ← registerService("database", connector);
    connected := true;
  }
}

```

```

protocol DBConnector roles client, server {
  actor skeleton = null;
  actor newClient;
  // Connect skeleton to server
  connectSkeleton(module server)
  if skeleton = null {
    skeleton := new DBSkeleton() as server;
    connect(skeleton) to server;
  }
  // Connect stub to client, alert skeleton of new stub
  requestStub(module requester) if true {
    newClient := new DBStub(skeleton) as client;
    skeleton ← addClient(newClient);
    connect(newClient) to requester;
  }
}

```

Figure 4: **Server and Connector Specification:** The server creates a protocol for connections, and registers the connector with the request broker. The connection protocol returns a skeleton to the server and processes connection requests from clients.

actor-type and *args* is passed as the set of initialization parameters when the new actor is created. After instantiation, the new actor is associated with the role *role-name*. Note that protocol actors may only be created in this fashion. Moreover, role assignments are permanent. Any actor created by a protocol actor is assigned to the role of its creator.

- `connect (args) to mod-ref`

Submits a connection request to the external module *mod-ref* with parameters *args*. Connection requests are always submitted asynchronously. As described in Section 2.1, if the connection is accepted, then any policy actor which is passed as a parameter is automatically admitted to the accepting module. Protocol actors may be admitted to multiple modules.

2.3 An Example Specification

As an example specification, consider a CORBA-like client-server architecture in which clients interact with a database server (see Figure 3). After the server is created it registers with a request broker. A connection is created between the client and server in three steps: (1) the client requests a connection from the request broker, who (2) forwards the request to the connection protocol, which (3) responds by submitting a stub to the client, thus allowing interactions with the server. We specify this behavior by creating a `DBServer` module and an associated `DBConnector` protocol. Figure 4 gives the code for the server and connector. For brevity, we have omitted the code for the request broker. We have also omitted any error handling code.²

²For example, verifying that a connection has been accepted. This would normally be accomplished by defining a method in the protocol

Upon creation, the `DBServer` module initializes its local state, creates a new instance of the `DBConnector` protocol, and sends a `connectSkeleton` message to request a skeleton from the new protocol. The `accept` rule defined by the server accepts the skeleton and forwards its address to the appropriate local actors which will handle incoming requests. All client requests will be routed through the `DBConnector` and delivered to the server through the skeleton. After the skeleton has been received, the server registers with the request broker by sending a `registerService` message.

The `requestStub` method defined by the `DBConnector` protocol is used to handle requests for client connections. Specifically, a client module is connected by consulting the request broker, which in turn invokes the `requestStub` method on the `DBConnector` protocol. The protocol creates a new stub actor, and issues a `connect` with the client module. After the client accepts the stub, interactions between the client and server modules will be routed through the client's stub, then to the server's skeleton, and finally to the server's internal actors. Note that all client stubs are assigned the `client` role, while the server's skeleton is assigned the `server` role. The purpose of assigning protocol actors to roles will become apparent in the next section where we discuss architectural policies.

3 Architectural Policies

An *architectural policy* defines a constraint over the manner in which a collection of actors invoke system services. For example, a load balancing policy might constrain the invocation of the `create` operation: each call to `create` may first require that a policy manager determine on which physical node the new actor should be created before servicing the

which is invoked by a module upon accepting a connection.

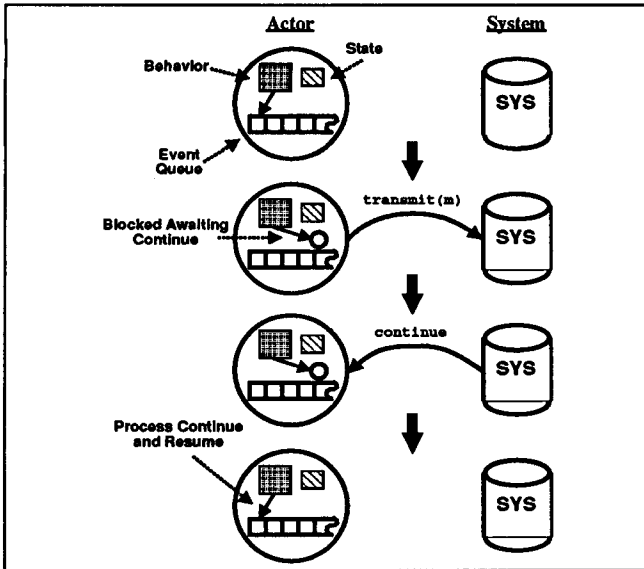


Figure 5: **Actor Signal Processing:** Actors request services by generating *signal* events. Send and create requests cause the actor to block until the request has been handled. Actors are resumed upon receiving an appropriate *notification* event.

request.

To support such policies, we augment actors with a meta-architecture which allows the customization of actor operations. A *policy actor* is an actor which is used to customize the operations of a module or protocol actor. A DCL policy defines a set of rules which are used to install and manage a collection of policy actors. In particular, policy actors are installed by invoking the install clauses of module or protocol specifications. The installation process admits policy actors as meta-level customizations of internal actors. Moreover, multiple policies may be installed on a single module or protocol. In this case, policy actors are “stacked” in the order of installation. The result is the composition of the behavior of each of the policies.

3.1 Meta-Level Customization

In order to customize actor behavior, we model actor interactions in terms of low-level event patterns. Specifically, an actor consists of a *behavior*, a *local state*, and an *event queue*. Actors compute by serially processing the events in their queue. In particular, an actor computation step consists of removing the first event from the event queue, changing the local state, and generating one or more new events. An actor generates a *signal* event in order to request a service. An actor receives a *notification* event as an acknowledgment that a previous signal has been processed.

Actor computation, as described in Section 1.1, is an abstraction over the processing of signals and notifications. In particular, the basic actor operations (*i.e.* *send*, *create*, and *ready*) are factored into signal-notification pairs. We treat certain signals as “resource requests”, and require the signaling actor to block until the resource is granted. Each blocking signal has a corresponding notification which may be used to resume a blocked actor (see Figure 7). For ex-

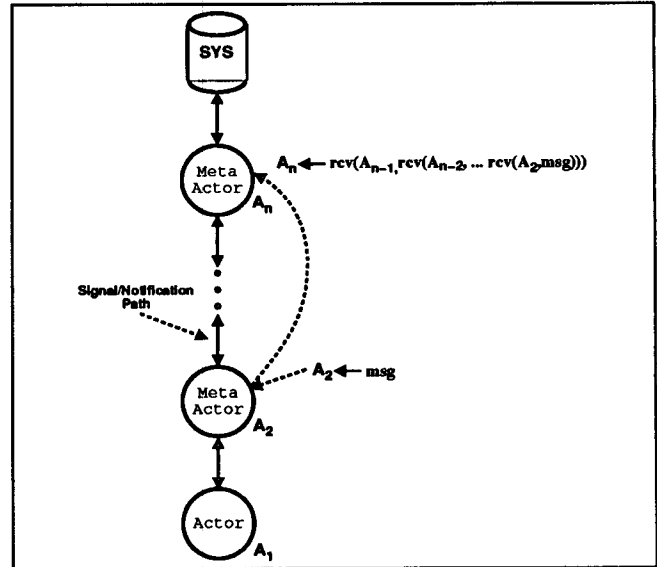


Figure 6: **A Meta-Level Stack:** Multiple policies are composed on a single actor by building a meta-level stack. Messages are redirected to the top actor in the stack and may be relayed down the stack to the appropriate target.

ample, a *send* operation blocks the requesting actor and generates a *transmit* signal directed to the underlying operating system (see Figure 5). The system might handle such an event by inserting the message on the network and generating a *continue* notification. The requesting actor resumes execution when the notification is received. Note that any other event received at the requester is queued until the *continue* notification has been processed. By default, signals are handled by a set of system services which perform the desired functionality. Meta-level customization is supported by allowing events to be processed by actors in place of system services.

A *policy actor* is an actor capable of processing signals and generating notifications. A policy actor which customizes another actor is referred to as a *meta-actor*. An actor customized in this fashion is referred to as the *base actor* relative to its policy actor. Once installed, a policy actor assumes responsibility for processing all signals generated by its base actor as well as generating notifications when necessary. Note that this includes the delivery of messages to the base actor: any messages targeted to the base actor are re-routed to the policy actor and annotated as *rcv* messages (see Figure 6). A policy actor may customize at most one other actor. Similarly, each actor may be customized by at most one policy actor.

Multiple policies may be installed on a single actor by allowing policy actors to customize other policy actors. In particular, multiple policies may be composed on a single actor by building a *meta-level stack*. Each policy actor is viewed as a customization of the actor immediately below it in the stack. Moreover, such customizations are transparent: a policy actor need not be aware that it customizes (or is customized by) another policy actor.

ACTOR TRANSITIONS		
	EVENT	BEHAVIOR
SIGNALS	transmit(<i>msg</i>)	Submit a message for transmission and block until a continue is received. The argument <i>msg</i> is a message structure which encapsulates the destination, method to invoke, and arguments of the message. The default system behavior is to send the message and send a continue notification to the signaling actor.
	ready()	Request the next available message for delivery. The default system behavior is to get the next available message and deliver it to the actor by generating a deliver notification. Note that this signal does not cause the caller to block.
	create(<i>b</i>)	Request the creation of a new actor and block until a newAddress is received. The argument <i>b</i> indicates the behavior of the new actor to create. The default system behavior is to create the new actor and deliver its address to the signaling actor by generating a newAddress notification.
NOTIFICATIONS	continue()	Resume an actor blocked on a transmit signal.
	deliver(<i>msg</i>)	Deliver a message to an actor. The argument <i>msg</i> is a message structure indicating the method and arguments to invoke on the resumed actor.
	newAddress(<i>a</i>)	Return the address of a newly created actor to an actor blocked on a create signal. The argument <i>a</i> indicates the address of the newly created actor.

Figure 7: **Actor Events:** Actor events are divided into *signals* and *notifications*. A signal represents a service request. A notification is an acknowledgement that a request has been serviced.

3.2 Policy Specification

A policy encapsulates a collection of actors and defines a set of rules for installing these actors as meta-level customizations (see Figure 8). As with modules and protocols, a policy defines a local state, an initialization section, and a set of methods. However, actors are not explicitly instantiated within policy methods. Instead, policy actors are created implicitly when the policy is installed.

A key challenge in applying a policy is to allow dynamic customization while respecting the integrity of module and protocol encapsulation boundaries. In particular, the internal composition of a module or protocol is not visible to external entities. To overcome this difficulty, policies are installed as either *contexts* or *roles*:

- **Context:** A policy applied to a **module** is called a *context* customization. In this form of customization, a uniform meta-actor type is instantiated and installed on each member of the module.
- **Role:** A policy applied to a **protocol** is called a *role* customization. In this form of customization, a uniform meta-actor type is instantiated and installed on each member of a role defined by a particular protocol.

Policy installations are initiated from within a method using the syntax:

```
install actor-type (args) on pol-target
```

where *actor-type* names the behavior of a policy actor, *args* parameterizes the behavior of each created actor, and *pol-target* represents the module reference or protocol role where the policy will be installed. The installation takes place only if the module or protocol represented by *pol-target* defines an *install* rule capable of accepting the request. For each actor *a* in *pol-target*, installation proceeds as follows:

1. A policy actor *m* of type *actor-type* is created with initial parameters *args*.

2. Actor *m* is admitted to the namespace of *pol-target*.

3. Actor *m* is installed as the meta-actor for *a*.

Installations are performed asynchronously. However, installations are serialized so that each policy actor is installed in a consistent fashion.³ A policy may be installed simultaneously as a context and a role. Moreover, a policy is not restricted to a single module or protocol. For example, a load balancing policy might be applied to every module or protocol in an architecture.

Because policy actors are installed in an encapsulated, but dynamically changing environment, we impose two additional constraints in order to ensure consistency:

- **Actor Creation:** Policy actors may only be created by installation. In particular, a create signal generated by a policy actor is treated as if the signal originated from the bottommost module or protocol actor in the meta-level stack. Moreover, the new actor is always admitted to the module or protocol represented by the bottommost actor.
- **Admission:** Any policy installed on a module or protocol role is automatically installed on any actor admitted after the initial installation. The installation is performed in the same order it was processed by the initial installation request, and each policy actor installed is parameterized using the same arguments as the initial installation.

The restriction on actor creation removes any ambiguity that may result when a create request is handled by a policy actor on behalf of its base actor.⁴ The restriction on admission ensures that each actor within a module or protocol role

³For example, the case where two separate policies are installed simultaneously on the same module will either correspond to the case where the first policy is installed in its entirety followed by the second, or vice versa.

⁴Without this restriction, it may be ambiguous as to which entity a new actor should be associated with: the underlying module or the installed policy?

```

policy ::= policy id {
    local-state
    [ init(args) { pol-action* } ]
    method*
}

method ::= [ local ] meth-name (args) {
    pol-action*
}

pol-action ::= local-state-assignment
| var-name ← meth-name (args)
| install actor-type (args) on pol-target

pol-target ::= mod-ref
| proto-ref <role-name>

```

Figure 8: **Policy Syntax:** A policy defines a local state, an initialization section, and a set of methods. The `install` keyword is used to install policy actors on modules or protocols.

has an identical meta-level stack. Recall that actor creation is handled as a special case of admission, so that any actor created by a module or protocol role will also be subject to any installed policies.

3.3 An Example Policy

As a simple example of a policy, consider the client-server architecture defined in Section 2. Suppose we desire secret interactions between the client and server. We can enforce this property by defining an `Encryption` policy which is applied to the `DBConnector` protocol. In particular, we might install an `Encrypt` policy actor on client stubs and a `Decrypt` policy actor on the server skeleton. Example code for these two policy actors is given on the top of Figure 9. Note that the `Encrypt` policy actor intercepts outgoing messages by defining a `transmit` method, while the `Decrypt` policy actor intercepts incoming messages by defining a `rcv` method. A policy which applies these actors is given on the bottom of Figure 9.⁵ The `apply` method is used to install the policy in two steps. First, a `Decrypt` policy actor is installed on the server role of the protocol. Because installation is asynchronous, we need to ensure that `Decrypt` has been installed before we begin encrypting client messages. The `Decrypt` policy actor calls the `setServer` method to alert the policy that it may safely install the `Encrypt` policy actor on clients. Note that the installation rules defined in the previous section ensure that any new actors admitted to the client role will automatically be customized by an `Encrypt` policy actor.

4 Conclusion

Modeling architectures using collections of actors provides a flexible mechanism for specifying arbitrarily concurrent, local computation while restricting remote communication

⁵Note that we have omitted the `install` statements which would need to be added to the protocol specification in Figure 4.

```

actor Encrypt(policy creator) {
    actor server;

    // Instantiated with name of server
    init (actor S) {
        server := S;
    }

    // Encrypt outgoing messages if they
    // are targeted to the server
    method transmit(Msg msg) {
        actor target = msg.dest;
        if (target == server)
            target ← encrypt(msg);
        else
            target ← msg;
        continue();
    }
}

actor Decrypt(policy creator) {
    // Alert creator when we have been instantiated
    init (policy creator) {
        creator ← setServer(self);
    }

    // Decrypt incoming messages targeted for
    // base actor (if necessary)
    method rcv(Msg msg) {
        if (encrypted(msg))
            deliver(decrypt(msg));
        else
            deliver(msg);
    }
}

policy Encryption {
    actor server;
    protocol target;

    // Install Decrypt on "server" role
    method apply(protocol T) if true {
        target := T;
        install Decrypt(self) on
            target<server>;
    }

    // Install Encrypt on "client" role
    method setServer(actor S) if true {
        server := S;
        install Encrypt(server) on
            target<client>;
    }
}

```

Figure 9: **Encryption Policy and Supporting Actors:** The `Encrypt` policy actor intercepts `transmit` signals and encrypts outgoing messages. The `Decrypt` policy actor intercepts messages targeted for the server (i.e. the `rcv` method) and, if necessary, decrypts an incoming message before delivering it. The `Encryption` policy coordinates the installation of `Encrypt` and `Decrypt` policy actors.

to adhere to a well-defined interaction mechanism. In order to provide a degree of modularity, collections of actors are encapsulated in restricted namespaces. The syntax of DCL is designed to allow extensions to these namespaces in order to form connections between architectural elements. Moreover, DCL abstractions are rule-based in anticipation of dynamic reconfiguration of distributed architectures. In order to allow policy management, we augment actors with a meta-architecture and provide the policy construct for coordinating the installation of customizations. Policies are composed by composing policy actors.

We have provided a high-level description of the semantics of DCL abstractions. A more complete semantics is based on a concurrent rewriting extension of actor semantics [12]. In particular, we extend this semantics by adding the notion of an *actor group*, which represents an encapsulated collection of actors. Composition such as that defined by the connection of protocols to modules is supported by allowing actor groups to overlap and exchange messages. Similarly, the composition of policies on other DCL elements is supported by allowing actor groups to exchange signals and notifications. The interactions between two groups defines an interaction semantics [13] which may be used to determine if two groups are compatible. If one group represents a module while the other represents a policy, then the interaction semantics may be used to verify conformance to an interface. Likewise, interaction semantics may be used to verify non-interference between a collection of policies. The interested reader is referred to [4] for a complete description of the semantics.

We are in the final stages of completing a prototype implementation which compiles DCL specifications into executable systems. Our prototype outputs a system of actors based on a literal interpretation of a DCL specification. The implementation is based on the Actor Foundry, a Java-based actor system and associated meta-architecture. This system is publicly available at <http://osl.cs.uiuc.edu/foundry>.

5 Acknowledgments

We thank past and present members of the Open Systems Laboratory who aided in this research. Extra gratitude is extended to Daniel Sturman whose comments were particularly useful in developing this work. We also thank the reviewers for their valuable comments. The research described has been made possible in part by support from the National Science Foundation (NSF CCR-9619522) and the Air Force Office of Science Research (AF BASAR 2689 ANTIC).

References

- [1] AGHA, G. *Actors: A Model of Concurrent Computation*. MIT Press, 1986.
- [2] AGHA, G., FRØLUND, S., KIM, W., PANWAR, R., PATTERSON, A., AND STURMAN, D. Abstraction and modularity mechanisms for concurrent computing. *IEEE Parallel and Distributed Technology* (May 1993).
- [3] ALLEN, R., AND GARLAN, D. Formalizing architectural connection. In *International Conference on Software Engineering (ICSE '94)* (1994), IEEE Computer Society, pp. 71–80.
- [4] ASTLEY, M. An actor semantics for component-based software. Tech. rep., University of Illinois at Urbana-Champaign, March 1998. Available at <http://osl.cs.uiuc.edu/~m-astle>.
- [5] CLEMENTS, P. C. A survey of architecture description languages. In *Eighth International Workshop on Software Specification and Design* (Paderborn, Germany, March 1996).
- [6] HOARE, C. Communicating sequential processes. *Communications of the ACM* 21, 8 (August 1978), 666–677.
- [7] HUTCHINSON, N. C., AND PETERSON, L. L. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering* 17, 1 (January 1991), 64–75.
- [8] LUCKHAM, D. C., KENNEY, J. J., AUGUSTIN, L. M., VERA, J., BRYAN, D., AND MANN, W. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering* 21, 4 (1995), 336–355. Special Issue on Software Architecture.
- [9] OBJECT MANAGEMENT GROUP. CORBA services: Common object services specification version 2. Tech. rep., Object Management Group, June 1997. Available at <http://www.omg.org/corba>.
- [10] SHAW, M., DELINE, R., KLEIN, D. V., ROSS, T. L., YOUNG, D. M., AND ZELESNIK, G. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering* (April 1995).
- [11] STURMAN, D. C. *Modular Specification of Interaction Policies in Distributed Computing*. PhD thesis, University of Illinois at Urbana-Champaign, May 1996.
- [12] TALCOTT, C. An actor rewriting theory. In *Workshop on Rewriting Logic* (1996), vol. 4 of *Electronic Notes in Theoretical Computer Science*.
- [13] TALCOTT, C. Interaction semantics for components of distributed systems. In *First IFIP workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS '96)* (Paris, France, March 1996).
- [14] THE JAVA TEAM. RMI specification. Available at <ftp://ftp.javasoft.com/docs/jdk1.1/rmi-spec.ps>.
- [15] TOMLINSON, C., CANNATA, P., MEREDITH, G., AND WOELK, D. The extensible services switch in Carnot. *IEEE Parallel and Distributed Technology* 1, 2 (May 1993), 16–20.
- [16] VAN RENESSE, R., BIRMAN, K. P., FRIEDMAN, R., HAYDEN, M., AND KARR, D. A. A framework for protocol composition in horus. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing* (August 1995).