CUSTOMIZATION AND COMPOSITION OF DISTRIBUTED OBJECTS: POLICY MANAGEMENT IN DISTRIBUTED SOFTWARE ARCHITECTURES

BY

MARK CHRISTOPHER ASTLEY

B.S., University of Alaska, Fairbanks, 1993
M.S., University of Illinois at Urbana-Champaign, 1996

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign,1999

Urbana, Illinois

Abstract

Research in software architecture has emphasized compositional development, where the computational aspects of a system are modularly separated from communication and coordination aspects. Typically, software architectures are factored into a set of *components*, which encapsulate computation, and *connectors*, which encapsulate interactions. In terms of design, development and debugging, this separation has several important advantages. In particular, by separating application code from the protocols used for interaction, software components may be independently developed and tested. Moreover, as requirements change, existing architectural elements may be modularly replaced by new elements with appropriate properties.

A fundamental problem with these abstractions is their interaction with "cross-cutting" architectural features such as heterogeneity, availability, and adaptability. Availability, for example, requires protocols that manipulate both communication and resources. Controlling architectural resources, however, requires access to the internal resource usage patterns of components and connectors. Unfortunately, current architectural abstractions have inflexible interfaces which obscure these patterns. This loss of information forces the implementation of such features to be hard-coded within architectural elements, eliminating many advantages of the modular approach.

In this thesis, we propose a model for distributed software architectures that exposes resource access in a modular fashion. Our model extends current architectural abstractions by providing a *meta-architecture* for customization. This meta-architecture augments the functional interface of architectural elements with an operational interface for controlling resources. We also develop a formal semantics which provides a foundation for reasoning about composition in the model.

As an instantiation of the model, we describe an architecture description language called the *Distributed Connection Language.* DCL allows the specification of distributed architectures which incorporate traditional elements (*i.e.* components and connectors) together with new elements, called *policies*, which specify resource constraints. We provide a Java-based implementation of DCL to demonstrate that the increased modularity of the approach does not entail prohibitive performance tradeoffs.

For Rufus the Cat, wherever ye may be...

Acknowledgements

> As a single balloon must stand for a lifetime of thinking about balloons, so each citizen expressed, in the attitude he chose, a complex of attitudes. One man might consider that the balloon had to do with the notion *sullied*, as in the sentence, *The big balloon sullied the otherwise clear and radiant Manhattan sky.* That is, the balloon was, in each man's view, an imposture, something inferior to the sky that had formerly been there, something interposed between the people and their "sky." But in fact it was January, the sky was dark and ugly; it was not a sky you could look up into, lying on your back in the street, with pleasure, unless pleasure, for you, proceeded from having been threatened, from having been misused. And to the underside of the balloon was a pleasure to look up into, we had seen to that, muted grays and browns for the most part, contrasted with walnut and soft, forgotten yellows. And so, while this man was thinking sullied, still there was an admixture of pleasurable cognition in his thinking, struggling with the original perception.
>
> – Donald Barthelme, "The Balloon", *Unspeakable Practices, Unnatural Acts*[1]

There are many people to thank for transforming an otherwise "sullying" experience into an "admixture of pleasurable cognition". First and foremost, I would like to thank my brother Scott and my parents for their continued support throughout the years. I would also like to thank Suzanne (Chicken Gruel Woman) for her love and support during the many ups and downs of graduate school.

My thesis committee - Gul Agha, Mehdi Harandi, Geneva Belford, Ralph Johnson, and Carolyn Talcott - have provided invaluable advice and comments on the work you see here. I would especially like to thank my advisor, Gul Agha, for providing encouragement throughout the years. Carolyn Talcott also deserves special mention for her help in developing the formal

[1] I am indebted to Dr. Joseph Dupras (a.k.a. Doctor Doom) in the English Department at the University of Alaska - Fairbanks for helping me track down this reference.

semantics in Chapter 4. Many of the results described here are a natural evolution of her foundational work. Needless to say, any remaining faults are solely my responsibility.

I would also like to thank the other members of the Open System Laboratory, both past and present, for their input and contributions. The “old gang”, Dan (I’m off coffee now) Sturman, Brian (Professor) Nielsen, Shangping Ren, Nalini Venkatasubramanian, Rajendra Panwar, Wooyoung Kim and Joonkyoo Yoo[2], all deserve special mention for providing comments on early versions of this work. The “new gang”, Thomas (Boot to the head) Clausen, Prasannaa (Mmmm, meat) Thati, James Waldby, Carlos Varela, Nadeem Jamali, and Reza Ziaei, deserve equal mention for providing comments and suggestions on early thesis drafts. The members of OSL also deserve credit for making the Actor Foundry project the success that it is.

Last but not least, I would like to thank Helena Mitasova, Bill Brown, and the other fine folks at the U.S. Army Construction Engineering Research Laboratory for their support throughout my graduate career.

[2] Sadly, Joon passed away in 1998.

Table of Contents

Chapter

List of Figures

Chapter 1

Introduction

1.1 Introduction

The term "middleware" has been used to describe cross-platform integration tools that support high-level distributed services such as remote interactions and fault-tolerance. In particular, it has been observed that, while distributed software may require complex interaction mechanisms, the implementation of these mechanisms need not be tied to the implementation of applications [53]. By separating application code from the protocols used for interaction, software components may be independently developed and tested. Moreover, as requirements change, existing architectural elements may be modularly replaced by new elements with appropriate properties.

The design methodology behind middleware is formalized by the notion of a *software architecture*. Software architecture factors a system into a collection of *components*, which encapsulate computation, and a collection of *connectors*, which describe how components are integrated into the architecture [50]. Such architectures are often described formally in terms of an *architecture description language* (ADL). An ADL provides linguistic abstractions that capture components and connectors, and support formal mechanisms for reasoning about composition. Recent ADL research has provided new insights in the areas of specification [7], verification [9] and prototyping [37]. Moreover, commercial middleware solutions such as the Common Object Request Broker Architecture (CORBA) [45] and Java's Remote Method Invocation (RMI) [58] embrace many aspects of the ADL approach. CORBA, for example, abstracts over the low-level remote procedure call (RPC) protocols required to link distributed objects. On the one hand, this allows designers to build distributed applications without regard for the specific low-level protocols in use. On the other hand, software architects may change these protocols (*e.g.* by creating new stub generation mechanisms) without changing application code.

Current ADLs emphasize the modular specification of coordination and communication. While this emphasis has demonstratable advantages for system development, a fundamental problem is loss of information engendered by ADL abstraction boundaries. This problem is particularly apparent when we consider the design of distributed software architectures. Specifically, distributed architectures introduce new concerns in system development:

- **Reliability:** Computer networks are inherently faulty and insecure. As a result, distributed applications may be exposed to intermittent loss of connectivity or to malicious attacks. Similarly, node failure is more probable in a multi-node configuration. [1] Thus, distributed applications may require replication or other forms of fault-tolerance in order to tolerate transient failures.

- **Heterogeneity:** Distributed applications may be deployed on heterogeneous hardware; different hardware may require different implementation techniques for application components. Moreover, component distribution may be a run-time rather than compile-time property; application components (*e.g.* Java applets) may need to adapt to different hardware at run-time.

Design constraints such as reliability and heterogeneity require a model of *resources* as well as a coherent interface for manipulating communication and coordination. For example, consider a critical server within a distributed application. We might increase the fault-tolerance of the server by adding a backup component. In order to utilize this approach, we must ensure

[1] The fact that multi-node configurations are more prone to failure than uniprocessor configurations can be seen by a simple probability argument: let p be the probability that a particular node will fail independently. Then $1 - p$ is the probability that the node will not fail. If there are N nodes, the probability that *none* of the nodes will fail is at most $(1-p)^N$. Since $(1-p)^N < (1-p)$ (because $p < 1$) we see that node failure is more likely in a multi-node configuration.

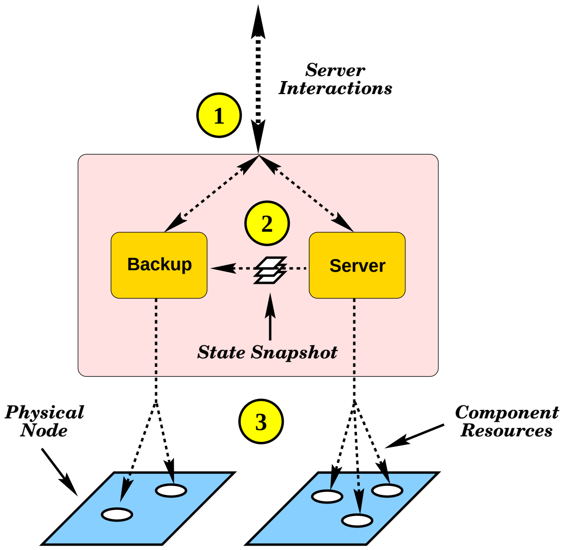


Figure 1.1: **Fault-tolerant Server:** The server is made fault-tolerant by three actions: 1) each server interaction is duplicated at the backup, 2) a state snapshot is periodically sent to the backup, and 3) the resources of the server and backup are separated to allow for independent failure.

that interactions and state snapshots are periodically recorded at the backup (see Figure 1.1). Moreover, we must ensure that resources associated with the server and backup are separated so that each may fail independently. However, without a modular mechanism for managing resources, we must implement such a policy by intermingling application and fault-tolerance code. This approach hinders development as new bugs may be introduced, and components cannot be tested independently. Thus, from a software engineering perspective, we desire modular mechanisms for extracting information such as state snapshots or resource placement, and allowing policies to be expressed separately as manipulations of these attributes.

In this thesis, we introduce new abstractions that capture architectural resources, and provide techniques for specifying policies over these resources. In effect, we augment the abstraction boundaries of current ADLs in order to support critical policies required in a distributed setting. Three main contributions characterize the approach: an architectural model for distributed software, a specification language and implementation, and a formal semantics.

Our architectural model is based on the Actor [2] model of computation. In particular, we view elements of a software architecture as encapsulated collections of actors, called *actor groups*. Through actor groups we realize the requirements necessary for a flexible model of distributed architectures: while internal actors are protected by the abstraction boundary provided by groups, composition mechanisms are provided that allow connections to be formed between groups. We impose a *meta-architecture* on actors to expose resources that are utilized by groups. The introduction of a meta-architecture provides a transparent, compositional mechanism for customizing the resource utilization of the actors within a group.

We develop the *Distributed Connection Language* as an Actor-based ADL that instantiates our architectural model. Linguistically, a DCL specification is rule based and reactive, specifying changes to an architecture in response to run-time interactions. DCL provides a compositional approach for architectural development: rather than hard-code customization and resource management within components and connectors, new abstractions called *policies* are used to modularly enforce such constraints.

While DCL allows for flexible system development, DCL abstractions have limited use if they can not be mapped to efficient implementations. In particular, a key performance issue is the overhead associated with modular composition versus hard-coded customization. Thus, as part of the development of DCL, we provide a reference implementation together with performance improvements that reduce overhead to less than 10%.

Finally, we provide a formal semantics for our architectural model. The purpose of this semantics is to provide a foundation for addressing composability issues in software architectures. In particular, we wish to provide formal techniques for determining whether or not the constraints defined by specific architectural elements are correctly enforced within the system. Through our semantics, we are able to define "weak" conditions for composability.

The remainder of this thesis is organized as follows. In the next section, we describe related work in software architecture and distributed systems. We then develop a model of distributed software architectures based on actors and their meta-level customization. Next, we describe DCL, a high-level specification language for distributed architectures. We describe the mapping of DCL specifications onto our architectural abstractions, as well as the performance characteristics of the approach. Finally, we conclude with a formal semantics of the model, as well as the derivation of composability conditions.

1.2 Related Work

The results developed in this thesis draw on related research in software architecture, reflective models, and more general research in distributed systems.

1.2.1 Architecture Description Languages and Software Buses

The predominant approach for specifying software architecture is by way of an *architecture description language* (ADL). Such a language provides abstractions that represent components and their interfaces, as well as separate mechanisms for connecting interfaces. An architectural specification is codified in terms of linguistic abstractions that define instantiations of components and connectors, and invoke binding operations to link interfaces. A similarly motivated approach is that of a *software bus.* A software bus provides a specification language that supports component descriptions but abstracts away most interconnection issues. Instead, the designer indicates which components should be connected to one another and the software bus automatically instantiates the appropriate connection mechanisms. Both approaches generally agree on component specification: a component is modeled as a discrete segment of code with a well-defined interface. Typically, interfaces are represented as function entry points and are invoked using traditional mechanisms such as procedure call or message passing. Moreover, composition is modeled uniformly in terms of hierarchical specifications of components that are represented as collections of sub-components.

While models of components are somewhat standard, the key contribution of recent work is modular connection mechanisms. In particular, these mechanisms are necessary for abstracting over component interconnection issues such as code heterogeneity, synchronization constraints, and fault-tolerance and security requirements. The manner in which connection mechanisms are handled marks the primary difference between ADLs and software buses. In an ADL, connection policies are represented explicitly by objects. On the other hand, in a software bus connection policies are represented implicitly as a feature included in the bus on a per-application basis; *i.e.* the set of components and their component interconnections is handled by a customized software bus instantiated explicitly for a particular application. These different choices for representing connection policies reflect different approaches for handling binding and customization issues, and have important implications in terms of how architectures may be customized dynamically. We describe representative ADLs and software buses and contrast their features in the remainder of this section.

1.2.1.1 UniCon

UniCon [50] is a high-level specification language that directs the compilation of an application by consulting the architectural specifications of pre-defined *components* and *connectors*. According to Shaw, a UniCon component "roughly corresponds to a compilation unit of conventional programming languages." A component specification consists of an interface and an implementation. An interface defines functions provided by the component as well as constraints that apply to the component's placement in an architecture. In particular, a component interface defines a set of *players* which represents the visible semantic elements through which components interact. A component implementation may be either composite, in which case it is defined in terms of a set of sub-components, or primitive, in which case it is defined by an executable object. In a composite component, interface players may be players provided by a sub-component.

A UniCon connector, on the other hand, is used to glue component interfaces together. A connector consists of a protocol and an implementation. A protocol is defined in terms of a collection of *roles* that designate the set of requirements for each participant in the connection. In effect, a role is a type specification that is used to determine if a component may participate as the given role in the protocol: the type specification of a component interface is a guarantee that the component adheres to a particular form of interaction. The implementation of a connector specifies the mechanisms used to carry out the interactions defined by the protocol. Currently, only primitive implementations are allowed. That is, composite connectors may not be specified.

The designers of UniCon have placed heavy emphasis on the implementation side of architectural specification. That is, component and connector specifications contain many detailed attributes that facilitate their use in specific contexts. As a result, UniCon specifications are at the mercy of low-level implementation concerns and it is difficult to abstract away a model of architectural structure. For example, `ProcedureCall` and `RemoteProcedureCall` are two instances of connectors. In order to use the `ProcedureCall` connector, a player must define a `RoutineDef` or `RoutineCall` attribute. On the other hand, in order to use the `RemoteProcedureCall` connector, a player must define an `RPCDef` or `RPCCall` attribute. However, the pattern of interactions is the same for either procedure call or RPC, only the context differs. Thus, it would seem logical that an ADL would abstract away these distinctions rather than embed them in component specifications.

1.2.1.2 Rapide

Rapide [36, 37, 38] is an object-oriented language designed for event-based prototyping of architectures of distributed systems. Rapide specifications are designed to allow system architects to test and verify architectures *before* implementation decisions are made. A Rapide architecture consists of a set of module specifications called *interfaces*. The behavior of the architecture is defined by a set of *connection rules* that represent direct communication between interfaces. The set of *formal constraints* for an architecture defines the permissible patterns of communication that may occur between interfaces. An interface defines a set of features that is an abstract description of the behavior of a module that conforms to the interface. Specifically, an interface defines behavior provided by a module as well as behavior required by the architecture that uses the module. Thus, in Rapide, no explicit components are present in an architectural specification. Rather, interfaces are surrogates for the actual modules used in an executable instantiation. The idea is to represent a type of behavior for simulation purposes: the actual choice of component is delayed until the architecture is implemented.

Rapide represents connections between modules in terms of event patterns among interfaces. The policies themselves are a combination of the connection rules and formal constraints for an architecture. Connection rules in Rapide represent direct links between elements of interfaces: events generated by one module are received at all modules which are connected by the connection rules. Connection rules may also be specified as patterns, which allows for the representation of dynamic interactions (*i.e.* where the set of participants may not be fixed at run-time).

While connection rules may be viewed as generators of event patterns, formal constraints restrict these patterns according to relationships between events. Formal constraints may either be placed in interfaces, in which case they may be specified in terms of abstract module behavior and restrict the local event patterns generated or received by the interface; or they may be placed in an architecture, in which case they restrict event patterns in the architecture as a whole. The formal constraints specified in an interface provide a "contract" describing the module's context in the architecture. This represents the simplest form of connection policy between two modules: that implied by their respective local constraints. These local connection policies may be augmented with additional constraints which define global properties.

The formal constraints specifiable in Rapide are motivated by the desire to ensure appropriate architecture-wide behavior, such as atomicity requirements or deadlock prevention. However, these architecture-wide constraints lack modularity and may easily interfere with one another. Thus, Rapide specifications are mainly useful as a design tool but do not provide a straightforward mechanism for mapping an architectural specification into an explicit implementation. In particular, while hierarchical refinement is well supported and interfaces correspond to actual components in an implementation, connection mechanisms are not encapsulated and therefore do not represent manipulatable elements in a realization of the architecture.

1.2.1.3 GenVoca

The GenVoca [16] system grew out of two separate environments for supporting the development of large hierarchically structured systems. Architectures for these systems are built from predefined system components with static interfaces. Components are organized into *realms* where implementations may vary but all components within a particular realm must implement the same interface. The interface shared by all components in a realm is specified in terms of a set of *operations* and *parameters*. Operations may be invoked by other components in an architecture and are used to access the services of the component. Parameters are place holders for other components and are used locally to obtain services from elsewhere in an architecture.

Connection policies in GenVoca are handled in a fashion similar to that of software buses. Specifically, components are organized into an architecture by *composition*, which is the process of resolving parameters in a component with instances of sub-components. Connection policies as separate, first-class entities do not exist. Rather, the process of composition matches interfaces according to type and the appropriate interaction mechanisms are automatically instantiated to handle the connection. A typical architecture is developed as a series of refinements starting with a high-level component which has several parameters. These parameters are filled with appropriate instantiations of sub-components (*i.e.* instances which satisfy the type of interface defined by the parameters), and the process continues until no further refinements are possible. These composition mechanisms have been used as the basis for domain-specific software system generators [15, 14].

The designers of GenVoca treat architectural specification in a domain of systems where architectural structure is relatively fixed and a rich space of components is available. As a result, the need for new interconnection mechanisms can only be satisfied by introducing new realms with new interfaces. Moreover, the notion of adapting a component to new contexts is not addressed. Rather, the designers envision a system in which components will be more intelligent and able to automatically adapt to future contexts. Pursuing this goal is the current research thrust of the project.

1.2.1.4 ABLE

The ABLE project at Carnegie Mellon has pursued software architecture from several different perspectives, culminating in the Wright [6, 7, 8, 9] architecture specification language and the ACME [24] architecture description interchange language. The Wright language embodies a more formal approach to architecture specification. In particular, behavioral aspects of an architecture are described by an extension of the language used in Communicating Sequential Processes (CSP) [26].

A Wright component is specified by an *interface* and a *computation*. An interface describes a fixed set of *ports* through which the component may participate in interactions. A computation describes what the component actually does. This specification is represented in terms of CSP events received at interface ports. Each port specification also defines a behavior which represents the behavior of the component relative to that particular port. For example, a simple filter component would define an input port with the behavior of receiving data, an output port with the behavior of sending data, and a computation with the behavior of computing an output based on the input. Each of these specifications is described in terms of CSP code fragments.

A Wright connector defines an interaction pattern between a set of ports. A connector specification consists of a set 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. When a connection is created, a behavior check is made to ensure that the behavior specification of a port satisfies the behavior specification of a role. This is done by proving a relaxed form of equivalence between the associated CSP code fragments. The glue of a connector is a complete behavioral specification of how events from one port are translated into events on another port.

Finally, a Wright configuration specifies a complete architecture in terms of a set of components and connectors to link their interactions. An important goal is to provide an explicit formal representation for all aspects of an architecture. Thus, the uniform model of computation provided by CSP is used as the notation for all architectural elements. As a result, architectures exhibit synchronous behavior. As noted in an early criticism of CSP [34], embedding synchronization constructs in communication constructs severely limits the ability to maximize concurrent resources.[2] As a result, a CSP based model may not be desirable for modeling the dynamic and highly asynchronous behavior of distributed systems.

[2] This is only half the story in a long-standing debate between proponents of synchronous and asynchronous communication. I only mention this criticism because it relates directly to the modeling of distributed systems (*cf.* [22, 63]).

1.2.1.5 Polylith and Aster

Polylith [27, 28, 47] and Aster [31, 30] are independently developed but similar paradigms which describe architectures in terms of a set of components, a set of connections, and a customized communication backbone, called a *software bus*, which all components use for interaction. A software bus encapsulates all the protocols necessary to manage interactions for a particular application and automatically invokes the appropriate protocols when components communicate. Binding components over a software bus consists of linking component interfaces directly. The software bus automatically determines a protocol to use for each interaction based on the properties of components. Customizing a software bus consists of modifying the implementation of protocols from which the bus is constructed.

Implicit connection mechanisms, such as software buses, have many desirable attributes. For example, architectural specification is greatly simplified as only connections between components need be indicated, rather than an explicit connection policy for each pair of components. Moreover, software buses are an appropriate abstraction for handling *vertical integration* [31] issues such as heterogeneity, availability and security. However, it is not clear how software buses can incorporate the dynamic interaction requirements of components. Specifically, it is not obvious how software buses may be made receptive to flexible communication topologies and per-interaction customization. A general problem is that software buses lack the fine-grained modular representation of object-based connectors. Therefore, it is not readily apparent how they may be customized for dynamic application-specific needs.

1.2.2 Modular Configuration of Distributed Systems

The extent to which heterogeneity is the norm rather than the exception in distributed systems has resulted in an emphasis on modular and customizable abstractions for building applications in these environments. Two approaches have been explored in contemporary work which differ mainly in the granularity of modularity and customization which is supported. Interaction specific or component based approaches emphasize customizable connections between individual components, while application based approaches specify customizations on a system-wide basis.

Customization of component interactions has been explored in the x-Kernel [29], Maud [4], and more recently in Horus [61]. The x-Kernel and Horus utilize protocol stacks to support customization. Each layer in the stack supports a static interface for interaction with the layers above and below it. The interface in these systems is fairly elaborate. Maud supports meta-level customization of protocols for fault-tolerance. In comparison to the protocol stack approach, reflection enables Maud to use a simple but flexible interface.

Application-oriented approaches, such as tool-kits, support a small set of protocols for interconnecting application components. For example, in transaction languages such as Avalon [20], the concept of nested transactions is used to structure distributed systems. Tool-kits, although lacking generality, are well suited to applications requiring only the protocols they provide. More recently, customization has been applied at the operating system level in the form of micro-kernels [1]. In an object-oriented system such as *Choices* [19], frameworks may be customized for a particular application. However, once customized, the characteristics may not change dynamically. In a similar manner, the x-Kernel allows customization of the message passing implementation. However, the modifications affect all components in the system.

1.2.3 Reflection and Language-Based Support

A final category in which concepts related to architectural configuration have been addressed is at the level of programming languages. Factoring out orthogonal application features such as synchronization constraints has been specified in approaches such as Synchronizers [23] and RT-Synchronizers [48, 49]. Similarly, modular specification of protocols for distributed interactions has been specified in terms of Communicators [53, 54]. A typical feature of these approaches is the introduction of separate language constructs that specify, for example, the coordination requirements between a group of objects.

A key problem with a strict language based approach is that new language constructs may have to be introduced in order to accommodate new architectures and applications. As Kiczales points out [33], the difficulty is that component implementation mechanisms are closed. That is, application specific client knowledge can not be utilized to yield efficient interaction mechanisms. Rather, interactions must adhere to mechanisms fixed in the implementations of particular components. Reflection in an object based system allows independent customization of each object. Reflection has been used to address many issues in concurrent systems. For example, the scheduling problem of the Time Warp algorithm for parallel discrete event simulation is modeled by means of reflection in [63]. Reflection has also been used explicitly to support parallel and concurrent language constructs [39]. Similarly, interconnection issues have been addressed reflectively in order to support flexible transaction models [13, 12]. Reflective frameworks for the Actor languages MERING IV and Rosette have been proposed in [21] and [60], respectively. In MERING IV, programs may access *meta-instances* to modify an object or *meta-classes* to change a class definition. In Rosette, the meta-level is described in terms of three components: a *container*, which represents the acquaintances and script; a *processor*, which acts as the scheduler for the actor; and a *mailbox*, which handles message reception.

Chapter 2

An Architectural Model for Distributed Software

We use the concept of *architectural context* as the basis for modeling distributed software architectures. The architectural context of a component consists of the resources it utilizes, and its connections with other components. Following Shaw and Garlan [51], we encapsulate computation within architectural *components.* In contrast, however, we define *connectors* as software elements which specify architectural context. Thus, we allow connectors to manage resources as well as manipulate connections between components. Connectors which manage resources may be used to capture more abstract architectural constraints such as replication and load management.

Three key features characterize our model:

- **Flexible Components and Connectors:** We provide an extensible model which allows the specification of multi-faceted elements within a uniform computing environment. In particular, we are able to capture common software abstractions such as sequential or multi-threaded execution; local, distributed or shared resources; and synchronous or asynchronous interactions. Moreover, the uniform nature of the model provides a grounded theoretical framework for reasoning about architectural interactions.

- **Encapsulation:** Abstraction boundaries are enforced by providing for component- and connector-based access control. In particular, internal resources are protected by a well-defined external interface. By enforcing encapsulation, we allow a compositional approach to software development: components and connectors may be independently developed and later incorporated into executable systems.

- **Composition Mechanisms and Reasoning:** Composition mechanisms are provided for integrating components and connectors while respecting encapsulation properties. Connectors which manipulate interactions are used to build connections between application components. Connectors which manage resources are used to enforce architectural constraints such as fault-tolerance, load management, and security. Moreover, our composition mechanisms provide a basis for reasoning about composition properties such as interface compliance and non-interference between resource management policies.

We use Actors [2] 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 [3, 53, 23]. Similarly, modern sequential languages are readily extended with the actor primitives (*cf.* [46, 59]).

We capture architectural context by incorporating a meta-architecture into the actor model. In particular, actor computation is represented in terms of low-level service requests which have a default system behavior. By allowing meta-actors to intercept these requests, we provide a mechanism for architectural customization. Specifically, the resource utilization of actors (*i.e.* patterns of service requests) may be controlled by a meta-actor which provides an alternative behavior for handling each request.

We represent components and connectors as collections of actors called *actor groups*. An actor group represents an encapsulation boundary which protects internal actors from external interactions: actors within a group may only exchange messages with other actors in the

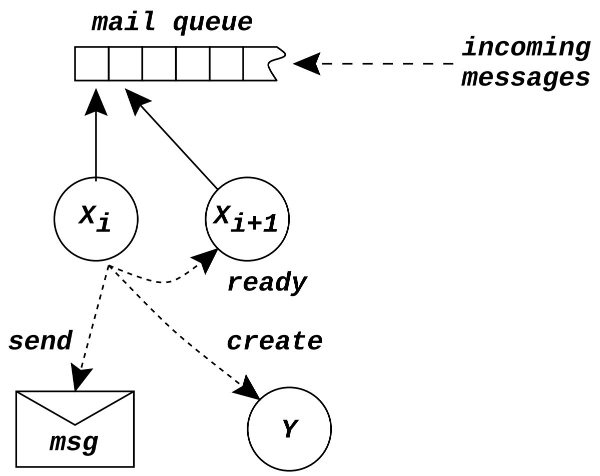


Figure 2.1: **Actor Model:** Actors are concurrent objects that interact via asynchronous messages.

same group. Composition operators are used to build connections between groups. Similarly, composition operators may be used to install meta-level customizations on group actors.

2.1 Actors

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 it is processing a message, there are three basic actions which an actor may perform that affect the computational environment (see Figure 2.1):

- *send* messages asynchronously to other actors;
- *create* actors with specified behaviors; and
- become *ready* to process 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.

In this thesis, we model actors as concurrent objects. That is, an actor consists of a private local state, a set of *methods*, and a globally unique name. Message passing is viewed as the asynchronous invocation of methods. We view the *send* and *create* operations as explicit requests, while the *ready* operation is implicit at the end of a method. That is, actors do not explicitly indicate that they are ready to receive the next message. Rather, the system automatically invokes *ready* when an actor method completes.

2.2 Meta-Level Customization

In order to capture architectural context, we view actor computation as an abstraction over low-level system interactions. Specifically, we define an actor to be composed of three attributes: a *behavior*, a *local state*, and an *event queue*. An *event* is a pair, (t, p), where t gives the *type* of the event, and p is an ordered list giving the *parameters* of the event. The type of an event is a constant chosen from a fixed set (*i.e.* there are a fixed number of event types). Actor computation is defined in terms of the processing of events; an actor computation step consists of removing an event from the event queue, changing the local state, and generating one or more new events.

Under this model, actors do not directly interact with one another. Instead, actors generate *signal* events which request the "system" to perform a particular action. In response to a signal, the system may service the request and generate a *notification* event which alerts the

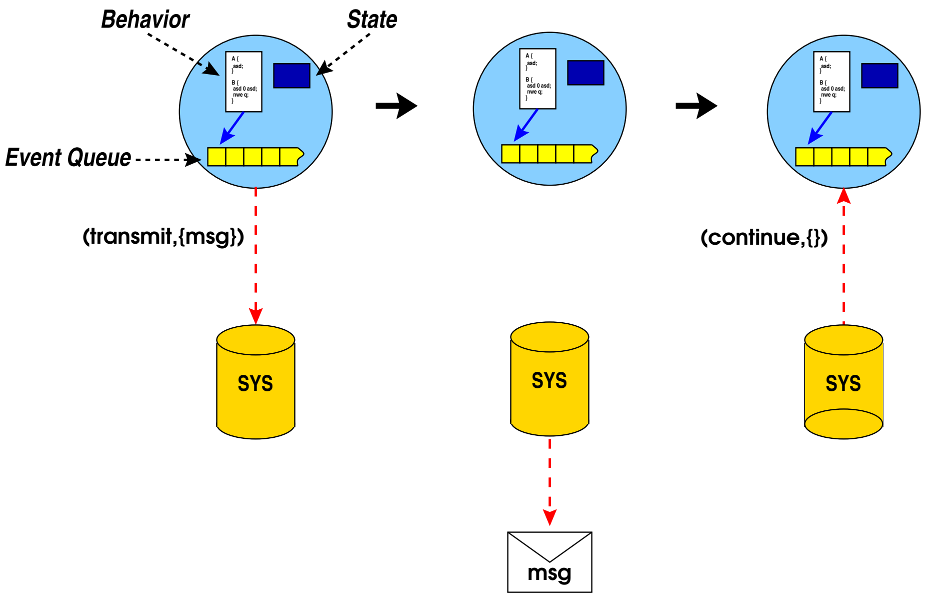


Figure 2.2: **Actor Signal Processing:** An actor requests a message send by generating a `transmit` signal. The system handles the signal by sending the message and generating a `continue` notification.

actor that its request has been processed. For example, an actor wishing to send a message may generate a "transmit" signal, (**transmit**, {`msg`}), where `msg` gives the message to send. The system handles the signal by sending the message and generating a "continue" notification, (**continue**, {}) (see Figure 2.2). On the receiving end, the system transforms the message into a "deliver" event which is eventually placed in the event queue of the appropriate actor.

By definition, signals represent resource requests and, therefore, always block the signaling actor until the resource has been granted or denied. An actor resumes processing when its event queue contains a notification corresponding to the signaled event. In particular, the desired notification is removed from the queue and processed as the next event. All other events are queued while the actor is blocked or processing another event.

Abstracting actor computation in terms of events decouples actor behavior from the servicing of requests. In particular, actors need not be specifically aware of the manner in which requests are serviced so long as event processing semantics are preserved. This independence

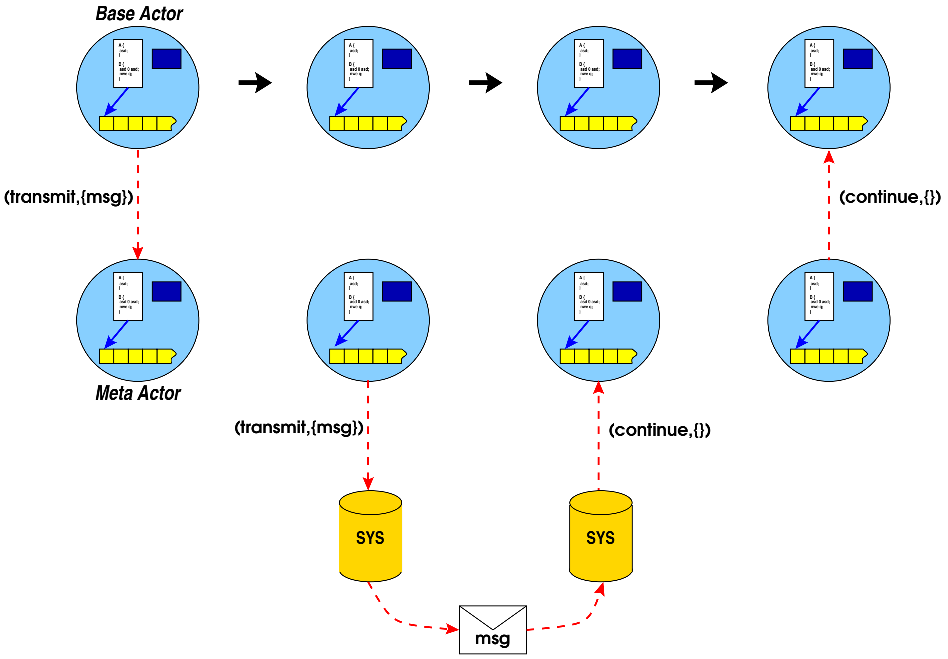


Figure 2.3: **Actor Customization:** A *meta-actor* customizes a *base-actor* by intercepting signals. The meta-actor generates a notification after each signal has been processed.

may be exploited to customize actor behavior. Specifically, we may transparently replace system behavior with equivalent mechanisms for servicing actor requests.

A *meta-actor* is an actor capable of processing signals generated by other actors. We use meta-actors as a mechanism for customizing actor behavior. For example, we may customize actor message passing by installing a meta-actor capable of handling `transmit` signals (see Figure 2.3). An actor customized in this fashion is referred to as the *base actor* relative to its meta-actor. Once installed, a meta-actor assumes responsibility for processing all signals generated by its base actor, as well as generating notifications when necessary.

Using the concept of meta-actors, a *meta-level architecture* is defined formally in terms of six attributes:

- **Events:** The set of *events* is finite and is divided into two non-overlapping subsets: *signals* and *notifications*.

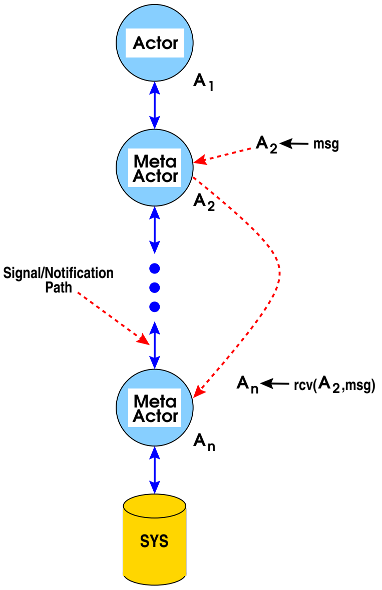


Figure 2.4: **A Meta-Level Stack:** Multiple customizations are composed on a single actor by building a meta-level stack. Messages are redirected (and annotated with "rcv") to the bottom actor in the stack and may be relayed up the stack to the appropriate target.

- **Blocking Relation:** A relation, $b \subset signals \times events$, defines the synchronization properties of signals. Specifically, if $(s, e) \in b$, then an actor blocked after generating signal s may be resumed upon receiving event e.

- **Installation by Creation:** Meta-actors may be installed only at their creation time. That is, an existing actor may not be used as a meta-actor unless created as such.

- **Propagation:** A *signal* generated by an actor is sent to its meta-actor or to the system. A *notification* generated by an actor is always sent to its base-actor, if it has one. Otherwise, the sending actor is considered "stuck" and may not process any further events.

- **One-to-One Installation:** Each base-actor may be customized by at most one meta-actor. Conversely, each meta-actor may customize at most one base-actor.

- **Message Delegation:** All *messages* targeted to base-actors are redirected to their meta-actors.

The set of events fixes the mechanism by which requests may be made and serviced within the model, while the blocking relation defines the synchronization properties of signals. In particular, the blocking relation specifies which events may be used to resume a blocked actor. Note that any event may be used to resume an actor (*i.e.* the set of resuming events is not limited to the set of notifications).

We insist on a one-to-one relationship between base-actors and their meta-actors in order to provide the most primitive model of customization.[1] However, multiple customizations may be applied to a single actor by building a *meta-level stack* (see Figure 2.4). That is, because a meta-actor is itself an actor, we may customize it by installing another meta-actor. Moreover, such customizations are transparent: a meta-actor need not be aware that it customizes (or is customized by) another meta-actor. Messages received by an actor in a meta-level stack are always delegated to the top of the stack so that a meta-actor always controls the delivery of messages to its base-actor. In particular, message delivery is handled by sending an appropriate notification to a base-actor containing the message to be delivered. We insist on "installation by creation" as a further simplification of the model.

Note that this model of meta-level customization is parameterized by the choice of events and the synchronization properties of signals. In the next section, we provide an instantiation of the model which captures the basic actor operations (*i.e.* **send**, **create**, and **ready**). In specific application areas, however, it may be desirable to instantiate the model with more specific events, or different synchronization properties. In Section 2.2.2, we consider other instantiations of the model.

2.2.1 Basic Actor Model

We factor each of the basic actor operations into a signal-notification pair (see Figures 2.5 and 2.6). While the `transmit` and `create` signals are handled in a straightforward fashion, the `ready` signal has a slightly different semantics. In particular, a meta-actor blocked on a

[1] Many-to-one relationships require that we define synchronization mechanisms among all the meta-actors customizing a particular base-actor. Fixing a particular synchronization paradigm is counterproductive to the flexibility we are trying to provide. Note, however, that many-to-one relationships may be simulated by providing an external "coordinator" actor which coordinates the behavior of individual meta-actors.

Operation	Signal	Notification
send	`transmit(msg)`	`continue()`
create	`create(beh)`	`newActor(a)`
ready	`ready()`	`deliver(msg)`

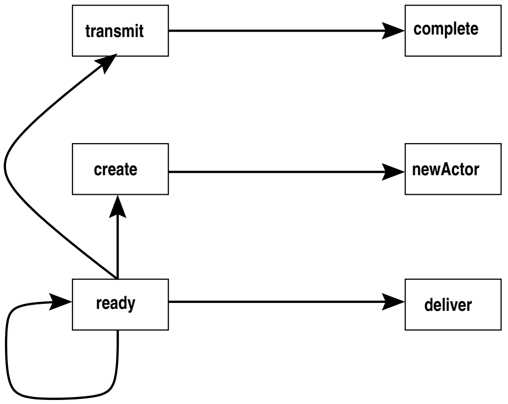


Figure 2.5: **Basic Actor Model Events and Synchronization:** The basic actor operations are factored into signal-notification pairs. As indicated by the arrows in the diagram on the right, the `transmit` and `create` signals block the caller until a `continue` or `newActor` notification is received, respectively. An actor blocked on `ready` may be resumed by a `deliver` or a signal from a base-actor.

`ready` may be resumed either by receiving a new message (*i.e.* a `deliver` notification) or by receiving a signal sent by its base-actor. This structure is necessary to avoid excessive meta-level interdependence. Specifically, if a meta-actor blocked on `ready` can only be resumed by a `deliver`, then the meta-actor can not process any requests from its base actor until it has received a new message from the system. Such a model is unnecessarily restrictive and prohibits many common architectural abstractions[2].

As an example of how we may customize actors under this model, consider the encryption of messages between a pair of actors. Figure 2.7 gives pseudo-code for a pair of meta-actors which may be installed on each endpoint. The `Encrypt` behavior intercepts outgoing messages by defining a `transmit` method. Within `transmit`, a message is encrypted before it is sent to its target. Note the use of `continue` to alert the base-actor that the `transmit` request has been serviced. The `Decrypt` behavior intercepts incoming messages by defining a `rcv` method. Recall that messages targeted for a base-actor are annotated with `rcv` and redirected to the

[2] An example of such an abstraction is a source actor: an actor which sends messages but never receives any. If such an actor is customized by a meta-actor which blocks on `ready` then, under the more restrictive semantics, the source actor may be blocked indefinitely.

Actor Transitions		
Event		Behavior
Signals	**transmit**(*msg*)	Submit a message for transmission and block until a **continue** is received. The argument *msg* 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.
	create(*beh*)	Request the creation of a new actor and block until a **newActor** is received. The argument *beh* 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 via a **newActor** notification.
Notifications	**continue**()	Resume an actor blocked on a **transmit** signal.
	deliver(*msg*)	Deliver a message to an actor. The argument *msg* is a message structure indicating the method and arguments to invoke on the resumed actor.
	newActor(*a*)	Return the address of a newly created actor to an actor blocked on a **create** signal. The argument *a* indicates the address of the newly created actor.

Figure 2.6: **Basic Actor Model Event Behavior:** Each signal has a default behavior corresponding to the actor semantics of the associated operation.

top of the meta-level stack. Within the `rcv` method, the message is decrypted and delivered by way of a `deliver` notification[3].

2.2.2 Model Extensions

An advantage of our model is that it is easily parameterized for specific application areas. For example, suppose we wish to design architectural policies for fault-tolerant systems. Specifically, suppose we desire the ability to make redundant backups of actors so that we may fail-over

[3] For ordering purposes, it might be necessary to wait for a `ready` signal before delivering a new message to the base-actor. In this example, however, it is not necessary to order incoming messages (actor message passing is assumed asynchronous), hence we may deliver new messages without waiting for a `ready`.

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

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

Figure 2.7: **Meta-Level Implementation of Encryption:** The `Encrypt` meta-actor intercepts `transmit` signals and encrypts outgoing messages. The `Decrypt` policy actor intercepts messages targeted for the receiver (via the `rcv` method) and, if necessary, decrypts an incoming message before delivering it.

when a fault occurs. To support this behavior, we re-parameterize (relative to Figure 2.6) the `create`, `newActor` and `ready` events as follows[4]:

- `create(`*beh*`,` *s*`,` *a*`)`: Request the creation of a new actor and block until a `newActor` is received. The argument *beh* indicates the behavior of the new actor to create. The argument *s* gives the initial state of the new actor. The argument *a* gives the desired address (*i.e.* actor name) of the new actor. If no specific address is required, then *a* may be set to the special symbol *nil*. The default system behavior is to attempt to create the new actor and return the resulting address in a `newActor` notification.

- `newActor(`*a*`)`: Resumes an actor blocked on a `create` signal. If the creation was successful, then *a* contains the address of the new actor. Otherwise, *a* contains the special symbol *nil*.

[4] Note that actor semantics are slightly altered under the new behavior of the `create` signal. That is, actor creation may not always succeed.

- `ready(`s`)`: Request the next available message for delivery. The argument s gives the current, consistent state of the requesting actor. The default system behavior is to get the next available message and deliver it to the actor.

```
actor Replicator(actor backup) {
    int processed = 0;
    int count = 0;
    boolean waiting = false;
    Queue mailQ;

    // Copy incoming messages to backup
    method rcv(Msg m) {
        // Send a stamped message to the backup
        backup ← rcvMsg(m, count++);

        // Queue until our base actor is ready
        if (waiting) {
                waiting = false;
                deliver(m);
        } else
                mailQ.enqueue(m);
    }

    // Forward state to backup and
    // deliver next message
    method ready(State s) {
        backup ← rcvState(s, processed++);

        if (!mailQ.empty())
                deliver(mailQ.dequeue());
        else
                waiting=true;
    }
}
```

```
actor Backup() {
    int count;
    State last;
    PriorityQueue unprocessed;

    // Receive new unprocessed message
    method rcvMsg(Msg m, int seq) {
        unprocessed.enqueue(m, seq);
    }

    // Receive new state
    method rcvState(State s, int seq) {
        last = s;

        Remove all message in "unprocessed"
        with sequence number less than seq
    }
}
```

Figure 2.8: **Meta-Level Implementation of Replication:** An instance of `Replicator` is installed on the actor to be replicated. An instance of `Backup` receives state snapshots from the `Replicator` so that it can assume the role of the replicated actor if a failure occurs.

Using this re-parameterized model, we may define a simple replication scheme based on the *primary-backup* protocol [18]. We implement primary-backup by defining a `Replicator` actor, which is installed on the actor to be replicated, and a `Backup` actor, which receives state snapshots captured by the `Replicator` (see Figure 2.8). The `Backup` records state snapshots so that it may assume the role of the replicated actor if a failure occurs.

We capture incoming messages in the `Replicator` actor by defining a `rcv` method. Upon receiving a new message, the `Replicator` sends a sequenced copy to the backup and either delivers the message to its base-actor, or queues the message for later delivery. The copies sent to the backup are received by the `rcvMsg` method and are sequenced to preserve their reception order at the `Replicator`. The `ready` method in the `Replicator` copies the current state of the replicated actor to the `Backup`. State snapshots are received by the `rcvState` method and are sequenced so that the `Backup` can determine which messages to discard from the local `unprocessed` queue. For the sake of brevity, we have omitted failure detection and fail-over code. However, these additions may be made in a straightforward fashion.

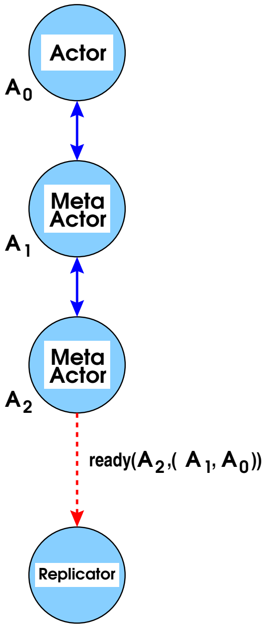


Figure 2.9: **Replication of a Meta-Level Stack:** In this case, the actor to be replicated consists of many actors organized into a meta-level stack. One solution is to change the semantics of `ready` so that state snapshots capture the entire state of the meta-level stack.

The example described above works well for simple meta-level customization. However, for more complicated scenarios it is necessary to make additional assumptions. For example, suppose the actor to replicate consists of several actors organized into a meta-level stack (see Figure 2.9). In this case, we need to capture the state of each actor in the stack. We may resolve this issue by implementing a solution similar to that described in [53]. Specifically, we require that each meta-actor adhere to the policy of constructing its state by incorporating a representation of the state of the actors beneath it. Although this is a slightly less general solution as each meta-actor must now pay attention to the state of the actors beneath it, it allows meta-actors the flexibility to determine what aspects of state are critical and what aspects are transient.

2.3 Actor Groups

We use encapsulated collections of actors, called *actor groups*, to represent the building blocks of distributed architectures. The purpose of an actor group is to isolate the behavior and resources of architectural elements behind clearly defined abstraction boundaries. We use collections, rather than individual actors, to allow for more flexible architectural structures. Specifically, while an individual actor may be used to model sequential computation, a collection of actors may be used to capture more expressive behavior such as multi-threaded components or distributed structures.

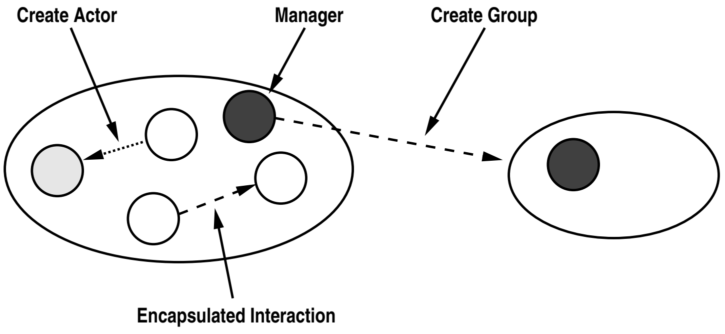


Figure 2.10: **Actor Groups:** Actors groups provide an encapsulated namespace. Actor creation and message passing is restricted within the group. Only managers may participate in external interactions.

Formally, an actor group is defined in terms of three attributes (see Figure 2.10):

- **Membership:** The collection of actors contained within a group is called the *membership* of the group. Each actor in the system is a member of at least one group. Moreover, actors may be members of multiple groups. That is, memberships may overlap. The group in which an actor is created is designated as the actor's *initial group*. The initial group of a new actor is the same as its creator.

- **Manager:** A single actor within each group is designated as the group manager. By convention, we refer to the *name* of a group as the actor name of its manager. A group manager has special privileges and is the only actor allowed to modify a group's membership, or create new external groups. However, group managers are subject to membership restrictions. Specifically, a group manager may only be a member of its initial group.

- **Namespace Encapsulation:** Actors may only interact with other actors in a common group. That is, actors in disjoint groups may not interact. Group managers have special privileges which allow them to receive messages from any other actor in a system, regardless of group membership. In particular, group managers may always interact with one another.

The membership of a group defines the computational behavior of a particular component of an architecture. Namespace encapsulation defines the abstraction boundary which protects internal computation from outside interference. In particular, actors within a group may not receive messages from external actors.

We view groups as representing the components and connectors of a software architecture (see Figure 2.11). For example, a group that represents a component encapsulates actors that perform computation, together with a set of actors for interacting with other groups (*i.e.* an interface). Similarly, groups that represent connectors encapsulate actors that are used as the endpoints for communication and coordination. However, a novel feature of our approach is the use of connectors as mechanisms for enforcing architectural policies. In this case, a group encapsulates actors that serve as meta-level customizations for actors in an external group. We separate meta-actors into separate groups to isolate components from the policies which govern them. In Chapter 4, we use this separation as a basis for deriving compatibility requirements between architectural structures.

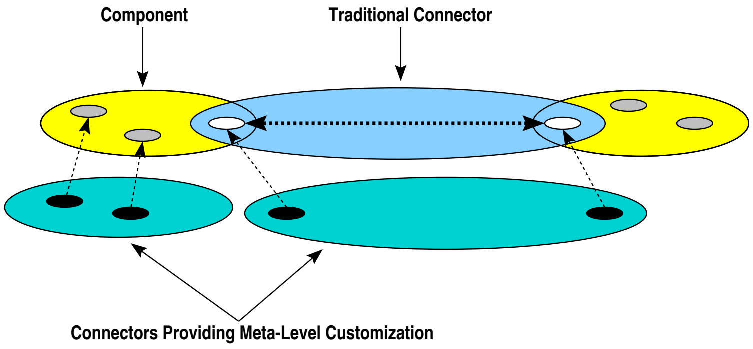


Figure 2.11: **Modeling Architectures with Actor Groups:** Actor groups are used to model components and connectors. Traditional architectural connection is achieved by groups which provide endpoints for communication and coordination. Architectural policies are enforced by groups which provide meta-level customization of other groups.

To model groups, we associate *membership* and *initial group* attributes with each actor. The membership attribute records the groups that an actor belongs to. The initial group attribute records the group that the actor was created in. Namespace encapsulation implies that two actors may only communicate if their membership attributes overlap[5]. Similarly, membership properties imply that the initial group attribute of a new actor is identical to the initial group attribute of its creator.

Group managers are the only actors allowed to relax the encapsulation properties of groups. Typically, encapsulation is weakened for one of two reasons: to allow local actors to *communicate* with actors in other groups; or to allow local actors to be *customized* by meta-level actors in other groups. Actors may belong to more than one group; group managers use *admission* operations to create such overlaps and establish the communication and customization relationships described above. For example, an *overlap* admission is used to build communication relationships. Similarly, a *customization* admission is used to build customization relationships. We describe admission in greater detail in the sections below.

[5] Because group membership is a dynamic property, it is possible that the sender and receiver of a message will be a member of a common group *after* the message has been sent. To avoid this ambiguity, we verify group membership at the time a message is sent. This constraint is formalized in Chapter 4.

2.3.1 Overlap Admission

An overlap admission allows a group manager to change its membership by adding an external actor. In terms of actor attributes, an overlap simply adds the admitting group to the membership attribute of the external actor. Once admitted, an actor may communicate with any other actor in the group.

Overlap admission allows otherwise disjoint groups to build conduits for interaction. Typically, a group manager will form an overlap by sending a request message to the manager of an external group. The contents of the message contain the actor(s) to be admitted, as well as any other parameters required to evaluate the request (*e.g.* keys for authentication). Once the admission has completed, the requesting manager may receive a notification indicating that the admission was successful. Note that, although the groups overlap, their encapsulation properties are still preserved. In particular, while it is possible for local actor addresses to be shared between the groups, only the overlapping actors may communicate with members of both groups. That is, non-overlapped actors are restricted from communicating directly.

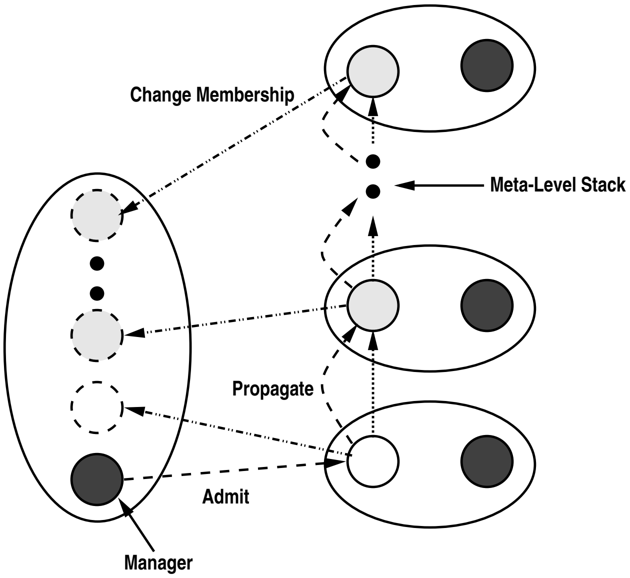


Figure 2.12: **Overlap of Meta-Level Stacks:** The overlap admission of a base actor is propagated to each actor in its meta-level stack. This ensures that the admitted actor may communicate locally.

While achieving overlap is a simple process for basic actors, note that an admitted actor may be customized by a meta-level stack. In this case, achieving overlap is a slightly more complicated procedure (see Figure 2.12). In particular, the meta-level stack of the admitted actor must *also* be admitted. This is necessary in order to allow the admitted actor to communicate with local actors. Specifically, if the admitted actor attempts to send a message using `transmit`, the message will actually be sent by the top actor in the meta-level stack. If the membership of this actor does not overlap with the membership of the destination, then the send will fail. To handle this case, the overlap admission is propagated up the meta-level stack before completing at the manager.

2.3.2 Customization Admission

The process of installing a meta-actor on a local group actor is called a *customization admission.* Recall that, because we wish to isolate components and policies, a meta-actor installed on a local actor must reside in a separate group. Moreover, the "installation by creation" property of the meta-architecture implies that the admitting group must create a new actor within an external group. Thus, the actor being admitted is the new meta-actor which was created in an external group by the installing manager. Although somewhat unorthodox, this behavior does not violate encapsulation properties: regardless of how the admission is performed, the new meta-actor is required to be a member of both groups once admission completes.

Customization admission allows actors of one group to serve as resource managers for actors of another group. We describe this relationship as enforcement of a policy (*i.e.* the collection of resource managers) over an architectural component (*i.e.* the collection of actors being customized). Typically, a group manager will install a customization by sending a request message to the manager of the group to be customized. The contents of the message contain a reference to the requesting manager, the actor behavior to be installed, as well as any other parameters required to evaluate the request. The receiving manager responds by creating and installing each of the requested meta-level behaviors (see Figure 2.13). The initial group attribute of each new meta-actor is set to the group of the requesting manager. The membership

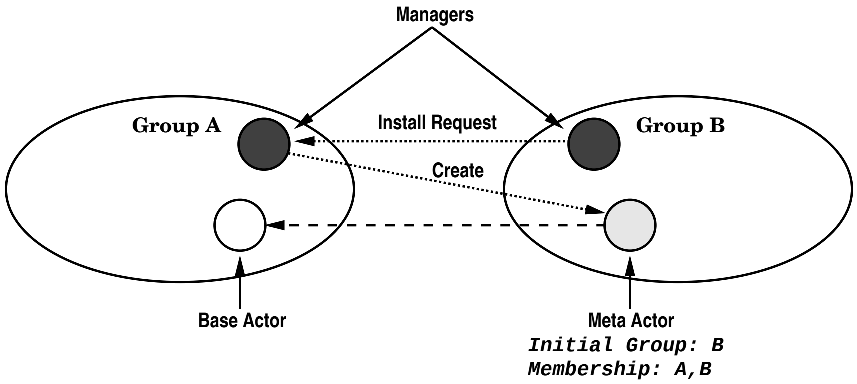


Figure 2.13: **Customization Admission:** The receiving manager (*i.e.* group A) creates a new meta-actor in the requesting group. The creating manager specifies the behavior of the new meta-actor, as well as the address of the local actor to be customized (some of this information may be contained in the install request). The new meta-actor has initial group equal to that of the requester (*i.e.* group B). The membership of the new meta-actor includes both groups.

attribute of each new meta-actor contains both the requesting and receiving group. The new meta-actors begin receiving base-level events once installation completes.

2.4 Summary

In this chapter, we provided an abstract description of our model. Through actor groups we realize the requirements necessary for a flexible model of distributed architectures. While internal actors are protected by the abstraction boundary provided by groups, managers may relax strict encapsulation in order to build connections with other architectural elements. Moreover, the introduction of a meta-architecture provides a transparent, compositional mechanism for customizing the architectural context of groups.

Admission operators provide composition mechanisms for distributed architectures. Admission by overlap is motivated by the need for forming connections between architectural structures. For example, a pipe and filter architecture may consist of actor groups representing filters with an additional group representing the pipe. Admission by overlap is used to connect filters to appropriate endpoints of pipes. Similarly, admission by customization is used to modify the architectural context of groups. In the pipe and filter example, it may be necessary to impose flow control over the pipe to compensate for different filtering rates. Flow control can be implemented by installing a separate group which uses meta-actors to customize message passing behavior of the pipe group. Admission by customization would be used to create and install each member of this group.

The use of both admission by overlap and admission by customization allows us to capture traditional architectural relationships (*i.e.* connectivity), as well as a new form of relationship that defines the customization of architectural contexts. In the next chapter, we consider an instantiation of our model in terms of the *Distributed Connection Language* (DCL), an architecture description language for distributed systems. DCL provides a concrete tool for building distributed software architectures, and provides a platform for reasoning about the applicability and performance of our approach.

Chapter 3

The Distributed Connection Language

The *Distributed Connection Language* (DCL) is an architecture description language for specifying distributed software architectures. An architectural unit in DCL is a linguistic abstraction over actor groups as defined in Chapter 2. The syntax of DCL is used to define the initial members of a group and the conditions under which admissions are performed. In particular, a basic specification in DCL consists of three 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.

- **Policy:** A policy defines a constraint over the manner in which a module or protocol invokes system services. Specifically, each actor within a policy is installed as a meta-level customization of an actor within a module or protocol.

In ADL terminology, a *module* is a component, while *protocols* and *policies* are connectors. Traditionally, components and connectors have a limited number of connection points and architectures are static structures fixed at specification time. In contrast, DCL abstractions are strictly dynamic: modules and protocols are rule-based, and DCL architectures are reconfigurable at run-time. Note that no generality is lost by restricting DCL to dynamic mechanisms. In particular, static architectural configurations may be viewed as an abstraction over the "bootstrapping" phase of a purely dynamic architecture[1].

Syntactically, the architectural elements of DCL (*i.e.* modules, protocols and policies) are specified by an "id" and a body. Element ids are used as type identifiers when new instances of the element are instantiated. The body of an element comprises two sections:

- **Local State:** Local state consists of a fixed set of local variables with simple types (such as integer, string, or array), or references to local actors and/or external modules, protocols, or policies. Elements of the local state are always passed by value when they are used in interactions with actors or external DCL elements. Within an actor or DCL specification, the special value self always refers to the name of the appropriate entity. An initialization section may be defined to initialize the local state when the element is instantiated. Note that an element may not participate in any interactions until its initialization code has completed.

- **Request Rules:** Request rules define the control interface of a DCL element. In general, a rule consists of a rule type, a caller id (*i.e.* a type specification for the sender of the event which triggers the rule), place holders for parameters, a boolean condition, and a rule body. An accept rule is a special rule defined within a module to accept a connection request from an external protocol. An install rule may be defined within a module or protocol and is used to allow policies to customize the internal actors of a module. A method is a general rule which may be defined within any DCL element. Methods are used for coordination among DCL elements.

[1] Static architectures also have the benefit of allowing compile-time type checking. However, we do not focus on compilation issues in this thesis.

Rules are matched in order of appearance. Only the first matched rule is invoked. If no rule is matched by a request, then the request is ignored[2]. 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 (*e.g. var-name* := *val*). New actors, modules, protocols, or policies 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, protocol, or policy and the new entity is created externally. *Var-name* is set to the name of the new entity after it has been created. That is *var-name* holds a value which may be used as a target for interactions.

The language by which actors are specified and the parameterization of the meta-level architecture are independent of DCL syntax. Note that many choices are possible for specifying actors [53, 35, 46]. For our purposes, we assume the existence of a separate language for specifying actors, together with a suitable instantiation of the meta-level architecture. We describe the syntax and semantics of DCL in the remainder of this chapter. Several example architectures are presented to drive the discussion. We conclude with a description of the implementation of DCL and a characterization of the performance aspects of the approach.

3.1 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 that is used to interact with the module. Module names are the only externally visible references in a module. While the actors within a module form a closed name space, any entity (internal or

[2] The decision to ignore unmatched requests is arbitrary and perhaps inappropriate in some cases where it might be useful to generate an exception message, for example, which is sent to the requester in response to an unmatched request.

otherwise) may use a module name as a target for interactions. Typically, module names are passed to external protocols or policies in order to initiate connections.

```
module      ::= module id {
                        local-state
                        [ init(args) { mod-action* } ]
                        accept-rule*
                        install-rule*
                        method*
                }

accept-rule ::= accept proto-id (args) if condition {
                        mod-action*
                }

install-rule ::= install policy-id (args) if condition {
                        mod-action*
                }

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

mod-action ::= local-state-assignment
             |  var-name ← meth-name (args)
             |  var-name := new actor-type (args)
```

Figure 3.1: **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. The syntax *rule** denotes the Kleene closure of a statement *rule*.

Figure 3.1 gives an abstract syntax for modules. As described above, local state consists of variables with primitive types or references to DCL entities. The init method is optional and specifies initialization code to be executed when the module is instantiated. The remainder of the syntax is defined as follows:

- accept *proto-id* (*args*) if *condition* {*mod-action** }

  Defines an accept rule. An accept rule matches a request if it originates from a protocol of type *proto-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 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 which is a member of the sending protocol, then this actor is automatically admitted to the module's name space before the body is evaluated (see Section 3.2).

- $\boxed{\textsf{install}\ \textit{policy-id}\ (\textit{args})\ \textsf{if}\ \textit{condition}\ \{\textit{mod-action}^*\ \}}$

  Defines an install rule. An install rule matches a request if it 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 3.3.

- $\boxed{[\textsf{local}]\ \textit{meth-name}\ (\textit{args})\ \textsf{if}\ \textit{condition}\ \{\textit{mod-action}^*\ \}}$

  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 parameters of the message and the rule body is evaluated.

- $\boxed{\textit{var-name}\ \leftarrow\ \textit{msg-name}\ (\textit{args})}$

  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 name space 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. We describe protocol and policy actors installation in more detail below.

3.2 Protocols

```
protocol      ::= protocol id [ role role-name* ] {
                            local-state
                            [ init(args) { proto-action* }]
                            install-rule*
                            method*
                  }

install-rule  ::= install policy-id (args) if condition {
                            proto-action*
                  }

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

proto-action  ::= local-state-assignment
                |  var-name ← meth-name (args)
                |  actor-name := new actor-type (args)
                            as role-name
                |  connect (args) to mod-ref
```

Figure 3.2: **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.

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 name space of each connected module. Protocol actors admitted in this fashion become members of both name spaces, 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 3.2). The connect action is provided to submit connection requests. Protocol syntax which differs from that of modules is defined as follows:

- $\boxed{\textit{actor-name} := \textsf{new}\ \textit{actor-type}\ (\textit{args})\ \textsf{as}\ \textit{role-name}}$

  Instantiates a new actor and assigns its reference to a local state variable. The type of the new actor is *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 this is the only mechanism for instantiating protocol actors from within protocol specifications. Moreover, role assignments are permanent. Any actors created by a protocol actor are assigned to the role of their creator.

- $\boxed{\textsf{connect}\ (\textit{args})\ \textsf{to}\ \textit{mod-ref}}$

  Submits a connection request to the external module *mod-ref* with parameters *args*. Connection requests are always submitted asynchronously. As described in Section 3.1, if the connection is accepted, then any protocol actor which is passed as a parameter is automatically admitted to the accepting module. Protocol actors may be admitted to multiple modules.

As with modules, policies may also be installed on protocols. A protocol accepts a policy by defining an *install* rule. In the case of protocols, however, policies are installed on individual roles rather than the protocol as a whole. This is done so that customizations may be isolated to specific endpoints of a protocol. We describe the installation process in more detail in the next section.

3.3 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 request.

A DCL policy encapsulates a collection of actors (called *policy actors*) and defines a set of rules for installing these actors as meta-level customizations (see Figure 3.3). 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.

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 single meta-actor type is instantiated and installed on each member of the module. Note that context customizations are only applied to actors created within the module. That is, admitted actors are not customized by context.

- **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.

As with modules and protocols, a policy defines a local state, an initialization section, and a set of methods. Although actors may be instantiated within policy methods, only the install keyword may be used to create actors which are used to customize module or protocol actors. The syntax of the install command is as follows:

install *actor-type* (*actor-args*) on *pol-target*(*rule-args*)

where *actor-type* names the behavior of a policy actor, *actor-args* parameterizes the behavior of each created actor, *rule-args* parameterizes the install request, 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 *actor-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. This is accomplished by multiple install commands, each with a different target.

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

- **Actor Creation:** Policy actors used for customization may be created only by installation. In particular, a create signal generated by a policy meta-actor is treated as if the signal came 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 created 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.

[3] For 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.

```
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 3.3: **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.

- **Multiple Customizations:** Multiple policies are enforced over a single module or protocol by using multiple instantiations of the install command. Installations are handled in the order they are processed by the install rule at the target module or protocol. The run-time ensures that each actor in the target has a consistent meta-level stack consistent with the installation order of policies.

By associating creation events with the bottommost actor in a meta-level stack, we remove 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 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. Finally, the restriction on multiple customizations provides a mechanism for asserting several policies over a single module or protocol.

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

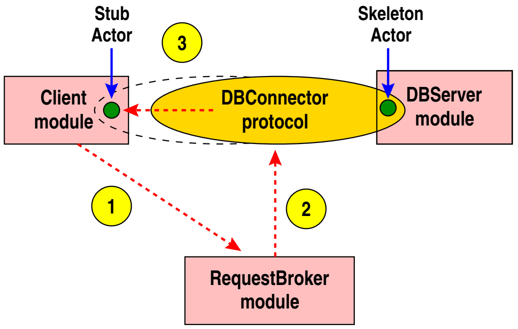
3.4 Example: CORBA-like Architecture

CORBA architectures are based on the client-server model. Specifically, servers are accessed through a *request broker* which facilitates connections between clients and registered servers. For example, suppose a client wishes to connect to a database server. The client first sends a request to the request broker. Assuming the database is registered, the request broker responds by creating a communication endpoint and delivering it to the client. Typically, this endpoint consists of a "stub" which implements the client end of the *remote procedure call* (RPC) protocol. Once the client receives the endpoint, it may make requests to the database.

With DCL, CORBA-like architectures may be modeled in a straightforward fashion. For example, we can model the client-server architecture described above by defining a `RequestBroker` module, a `DBServer` module and an associated `DBConnector` protocol. A connection is created with the server in three stages (see Figure 3.4): (1) the client requests a connection from the request broker, which (2) forwards the request to the connection protocol, which (3) responds by submitting a stub to the client, thus allowing interactions with the server. Figure 3.5 gives the code for the server and connector. For simplicity, we have omitted any error handling code such as code to verify that a connection has been accepted. This could be done by defining a method in the protocol which is invoked by a module upon accepting a connection.

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.

Clients request connections by invoking the `connectClient` method in `RequestBroker`. The `RequestBroker` forwards the request by looking up the appropriate protocol and sending a `requestStub` request. In the `DBConnector` protocol, the `requestStub` method creates a new stub actor, and issues a `connect` with the client module. After the client accepts the stub,



```
module RequestBroker {
  hashtable servers;
  protocol link;
  method registerService(string name,
                              protocol link) {
    servers.put(name, link);
  }
  method connectClient(module client,
                              string service) {
    link := servers.get(service);
    link ← requestStub(client);
  }
}
```

Figure 3.4: A CORBA-like Client-Server Architecture: Left: The `RequestBroker` forwards connection requests to the `DBConnector` instance associated with the server. Right: The `RequestBroker` specification defines rules for registering servers and accepting connection requests.

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.

3.4.1 Customizing Server Connections

CORBA architectures may be customized by changing the implementation of the request broker, or providing alternate implementations of connectors. This approach complicates system development as modifying existing components may introduce new bugs or architectural incompatibilities.

DCL policies offer a simpler approach. For example, suppose we wish to encrypt interactions between the client and database. We can enforce this property by defining an `Encryption` policy which is applied to the `DBConnector` protocol. In particular, we may use the `Encrypt` and `Decrypt` meta-actors defined in Section 2.2.1, where we slightly modify the `Decrypt` behavior so that it is instantiated with the address of its creator. When an instance of `Decrypt` is created, it forwards its name to its creator by calling the `setServer` method (the purpose

```
module DBServer {
    boolean connected = false;
    module broker;
    protocol connector;

    init(module rBroker) {
        Create internal resources;

        // Save reference to request broker
        broker := rBroker;

        // Create connection, request skeleton
        connector := new DBConnector();
        connector ← connectSkeleton(self);
    }

    // Only accept one skeleton
    accept DBConnector(actor skeleton)
      if !connected {
        Forward skeleton to 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, alert skeleton
   requestStub(module requester) if true {
      newClient := new DBStub(skeleton)
         as client;
      skeleton ← addClient(newClient);
      connect(newClient) to requester;
   }
}
```

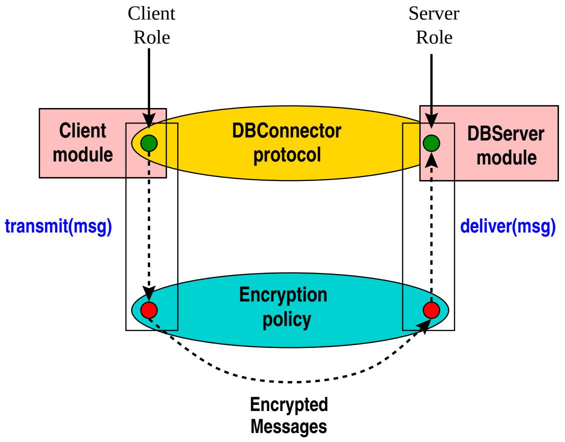
Figure 3.5: **Server and Connector Specification:** The server creates a protocol for connections, and registers the protocol with the request broker. The connection protocol returns a skeleton to the server and processes connection requests from clients.

for this behavior is to establish an appropriate receiver for the `Encrypt` behavior). We install instances of `Encrypt` on client stubs, and instances of `Decrypt` on server skeletons. Note that the `Encryption` policy may only be installed if the an install rule has been defined in `DBConnector`. Thus, a rule of the form:

```
install Encryption() if true {}
```

must be added to `DBConnector` before the installation will be permitted. More complicated rules may be specified to model specific behavior. For example, policies may be required to present a "signature" or other form of authorization before the installation is allowed. This condition could be enforced with an appropriate rule argument and corresponding rule condition.

The `Encryption` policy is given in Figure 3.6. Typically, such policies will be installed by an external entity which first creates an instance of the policy and then calls an appropriate



```
policy Encryption {
   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 {
      install Encrypt(S) on target<client>();
   }
}
```

Figure 3.6: **Encryption Policy:** The `Encryption` policy coordinates the installation of `Encrypt` and `Decrypt` meta-actors. Once installed, all interactions between clients and the server are encrypted.

policy method to initiate the installation. In the case of `Encryption`, 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.

3.4.2 Controlling Server Resources

While it is possible to customize interactions in CORBA (albeit with some difficulty), it is not possible to customize the resource usage of CORBA clients without modifying the clients themselves. However, in a CORBA-like architecture specified in DCL, we may make such customizations by applying policies to modules. For example, suppose that the server executes on a cluster of workstations. Suppose further that we wish to load balance the server's resources to increase performance. We might enforce such a constraint by load balancing the actors created by the server. That is, we control the initial placement of each actor in the server.

```
actor LoadBalance {
   actorclass nextType;
   policy mgr;

   method init(policy m) {
      mgr := m;
   }

   method create(actorclass type, string host) {
      nextType = type;
      mgr ← requestMachine(self);
   }

   method rcvMachine(string newHost) {
      newAddress(create(nextType, newHost));
   }
}
```

```
policy RoundRobin {
   string hosts[] = { a list of hosts };
   int nextHost = 0;
   int numHosts = hosts.length;

   method installLoadBalance(module mod) {
     install LoadBalance(self) on mod();
   }

   method requestMachine(actor caller) {
     caller ← rcvMachine(hosts[nextHost]);
     nextHost = (nextHost + 1) % numHosts;
   }
}
```

Figure 3.7: **Load Balance Policy:** Instances of `LoadBalance` are installed on the server by the `RoundRobin` policy. Once installed, each `create` request is forwarded to the policy in order to determine where to create the new actor.

Figure 3.7 gives the specification of a `LoadBalance` meta-actor, and a `RoundRobin` policy. For this example, we assume that the `create` meta-level signal is parameterized with both an actor behavior, and the name of the host where the new actor should be created. The `RoundRobin` policy installs an instance of `LoadBalance` on every actor in the server. The `LoadBalance` meta-actor intercepts create requests and forwards them to the `requestMachine` method in `RoundRobin`. This method determines where the new actor should be created and sends the location back to the requesting actor. Once the new location has been received, the new actor is created and a reference is returned to the requesting base actor.

As with the encryption example above, we require an “install” rule defined within the `DBServer` module. However, once installed, the `RoundRobin` policy is completely transparent to the internal server actors.

3.5 Example: High-Availability Server

In the previous section, we described the customization of a CORBA-like architecture defined in DCL. As a more complicated example, consider a bug-tracking database. We attach an

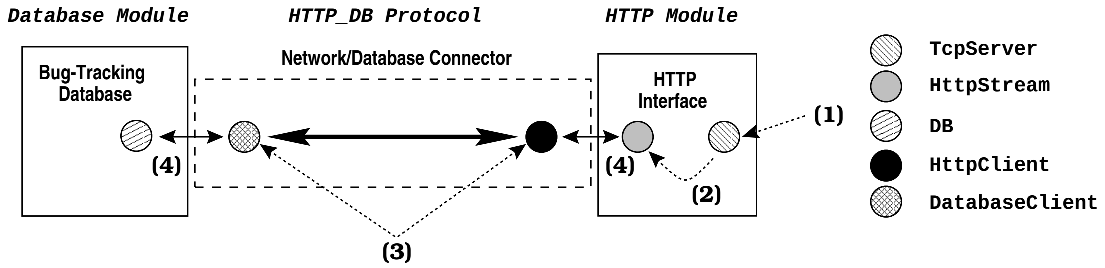


Figure 3.8: **A Bug-Tracking Database:** An HTTP interface module provides access to a database storing bug reports. A network-to-database connector translates HTTP requests into document requests at the database.

HTTP front end to the database in order to make bug-tracking information available via the web (*e.g.* using a web browser). The architecture of this system may be defined in terms of three components:

- **Bug Database:** The bug database stores bug reports. Clients use the *get* method to request a particular bug report, and the *put* method to insert a new bug report or make a modification to an existing report.
- **HTTP Interface:** The HTTP interface is used to make the database accessible to remote clients. In particular, the interface listens for new TCP/IP connections on port 80 (*i.e.* HTTP), and creates an appropriate link to the database for each new connection.
- **Network to Database Link:** The HTTP interface and database are linked by a protocol which translates HTTP requests into appropriate invocations of *get* and *put* on the database.

The DCL specification for this application consists of a `Database` module, an `HTTP` module, and an `HTTP_DB` protocol (see Figures 3.8 and 3.9). The `HTTP_DB` protocol is initialized with a reference to the `Database` module, while the `HTTP` module is initialized with a reference to the `HTTP_DB` protocol.

During initialization, the `HTTP` module creates a `TcpServer` actor to listen for new connections. Upon receiving a connection request, the `TcpServer` actor creates a `HttpStream` actor to handle the new connection (stages (1) and (2) in Figure 3.8). The address of the HttpStream actor is forwarded to the handleNewConnection method defined in the HTTP module (the new actor is bound to the newStream parameter). This method links to the new connection to the database by calling the addConnection method in HTTP_DB. Note the use of the local keyword to protect handleNewConnection from non-local requests.

The init method of HTTP_DB creates a DatabaseClient actor to serve as a communication endpoint for the DB module. The connect statement in init ensures that the DB module receives the endpoint during protocol initialization.

The addConnection method builds a new connection to the database each time it is called. In particular, a new HttpClient actor is created to serve as a communication endpoint for the HTTP module (stage (3) in Figure 3.8). The connect statement in addConnection submits the HttpClient actor to the HTTP module. The accept method in HTTP handles the final stage of the connection by sending the address of the appropriate HttpStream actor to the admitted HttpClient actor. Once connected, the DatabaseClient and HttpClient actors coordinate to translate HTTP requests into database requests, and vice versa (stage (4) in Figure 3.8).

This architecture is designed to be deployed on a single server. However, if the bug database holds critical information, it may be necessary to increase the availability properties of the system. In particular, suppose we have access to a cluster of three workstations. We may consider two modifications to this architecture in order to increase availability (see Figure 3.10):

- **Replicated Database:** Replicated copies of the database may be maintained in order to resist transient failures as well as provide more rapid access to clients.

- **Load Balance Connections:** In order to increase throughput, we may want to selectively route incoming connections so that they are balanced across the available hardware.

We implement this modified architecture using the BalanceConnections and SequenceRequests policies (see Figure 3.11[5]). We designate one of the replicas as the "leader" and install the

[5] We omit the description of the BalanceConnections policy. This policy simply installs the Rerouter actor on the "leader" HTTP module.

```
module HTTP {
   actor tcpServer;
   queue NC;
   protocol dbConnector;

   // Initialize Module
   init(Protocol C) {
     dbConnector = C;
     NC = new Queue();

     // Create actor for HTTP connections
     tcpServer = new TcpServer(80, self);
   }

   // Called by tcpServer to add new connection
   local handleNewConnection(Actor newStream)
     if true {
     // Save actor for new connection
     NC.enqueue(newStream);

     // Create new protocol link
     dbConnector <- addConnection(self);
   }

   // Accept a new HTTP_DB connection
   accept HTTP_DB(Actor httpClient)
     if !NC.empty() {
     // Send address of local stream
     httpClient <-
        setStreamActor(NC.dequeue());
   }
}
```

```
protocol HTTP_DB role HttpEndpoint, DBEndpoint {
   Actor HttpActor, DBActor;
   Module dataBase;

   // Build initial connection to DB
   init(Module DB) {
      dataBase = DB;
      DBActor = new DatabaseClient()
        as DBEndpoint;
      connect(DBActor) to dataBase;
   }
   // Build new connection to database.
   addConnection(Module caller) if true {
      // Create client endpoint for connection
      HttpActor = new HttpClient(DBActor)
        as HttpEndpoint;

      // Submit endpoint to HTTP
      connect(HttpActor) to caller;
   }
}
```

Figure 3.9: **Bug Database Specification:** The database and HTTP interface are specified as DCL modules. The Network/Database connector is specified by the HTTP_DB protocol. For the sake of brevity, we omit the specification for the database. Note that all we require from the database is an accept rule for accepting the DatabaseClient actor created in HTTP_DB.

BalanceConnections policy on the HTTP module at that replica[6]. The SequenceRequests policy is installed collectively on each of the HTTP_DB protocols.

The BalanceConnections policy uses a Rerouter actor to customize the message passing behavior of the TcpServer actor inside the HTTP module. In particular, the Rerouter intercepts handleNewConnection messages, and determines whether they should be handled locally or forwarded to another HTTP module. In the latter case, the connection is forwarded by sending a

[6] Ideally, the leader should be a separate module residing on a router. For simplicity, however, we arbitrarily choose one member of the cluster.

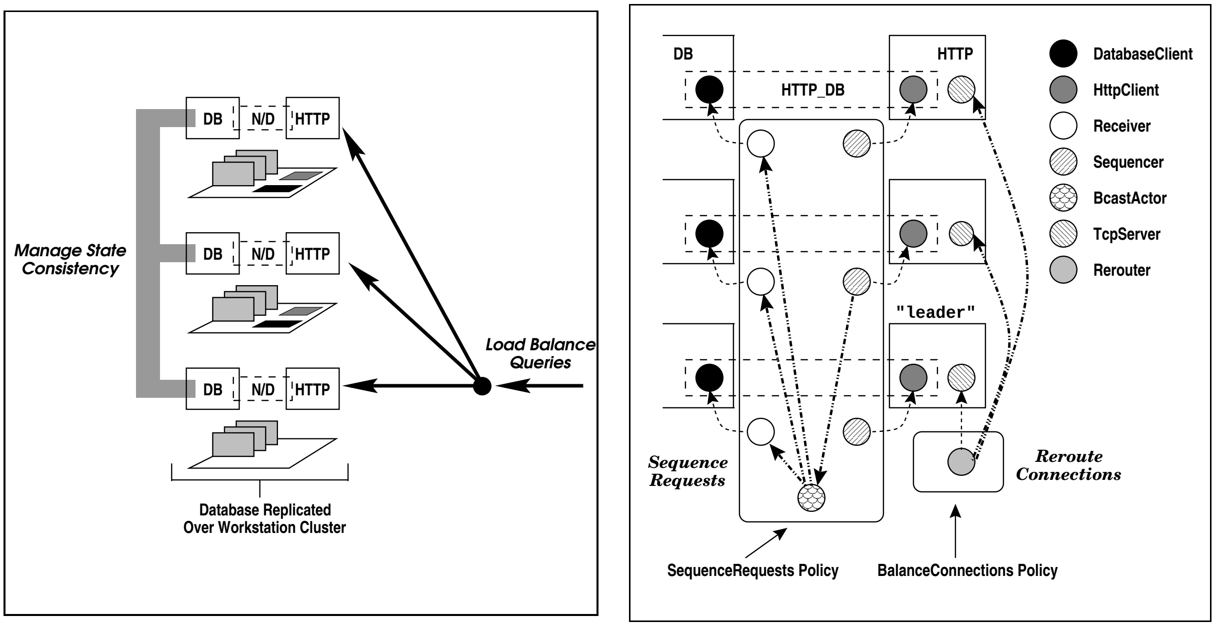


Figure 3.10: **Cluster Deployment:** On the left: proposed changes to create a high-availability server. On the right: architectural changes to support the new server.

"Moved Temporarily" reply to the HTTP caller. At present, the `Rerouter` actor uses a simple round-robin scheme to allocate requests.

The `SequenceRequests` policy maintains the consistency of the replicas by ensuring that all requests are processed in the same order at each replica. This is done by installing a `Sequencer` actor on each member of the `HttpEndpoint` role, and a corresponding `Receiver` actor on each member of the `DBEndpoint` role. The `Sequencer` actor intercepts request messages and invokes the `receiveRequest` policy method to assign a sequence number to the request. The request is then forwarded to each `Receiver` actor, which delivers requests according to their sequence number. Replica consistency is guaranteed because requests are serialized at each `Receiver`. In particular, all requests will be processed in the same order at each replica.

As in the previous example, we must add appropriate install rules to the `HTTP` and `HTTP_DB` specifications before our policies may be installed. For example, we might add the simple rule:

```
install SequenceRequests() if true { }
```

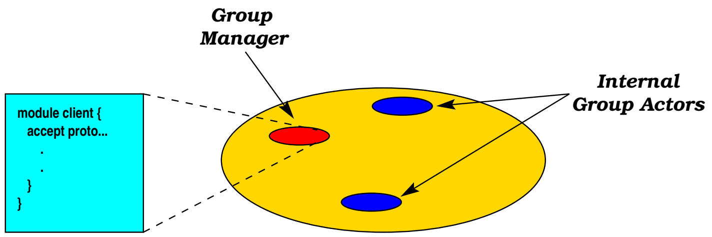
to the `HTTP_DB` specification.

```
policy SequenceRequests {
  int seqNum;
  Set receivers;
  Actor caster;
  Actor target;
  // Install policy on each protocol
  init(Protocol C1, Protocol C2, Protocol C3) {
    seqNum = 0;
    receivers = new Set();
    caster = new BcastActor();
    // Sequencer sequences new requests
    // Receiver delivers requests in order
    install Sequencer(self) on C1<HttpEndpoint>();
    install Sequencer(self) on C2<HttpEndpoint>();
    install Sequencer(self) on C3<HttpEndpoint>();
    install Receiver(self) on C1<DBEndpoint>();
    install Receiver(self) on C2<DBEndpoint>();
    install Receiver(self) on C3<DBEndpoint>();
  }
  // Register new database receiver
  local registerReceiver(Actor who) if true {
    receivers.addElement(who);
  }
  // Sequence request, broadcast clients
  local receiveRequest(HttpRequest req, Actor R)
   if true {
    // Forward request to intended receiver,
    // broadcast to other DB clients,
    // and increment sequence number
    R <- request(req, seqNum);
    caster <- sendCopy(req, seqNum,
                          receivers - R);
    seqNum++;
  }
}
```

```
actor Sequencer {
  policy ourMgr;
  // Save reference to our policy mgr
  init(Policy M) { ourMgr = M; }
  // Intercept message sends
  method transmit(Msg msg) {
    if (msg.method == "newRequest")
      // If request, then send to be sequenced
      ourMgr <- receiveRequest(msg.arg[0],
                                  msg.dest);
    else
      // Otherwise, send to destination
      msg.dest <- msg;
    continue();
  }
}

actor Receiver {
  policy ourMgr;
  int seqNum = 0;
  PriorityQueue sequenced = new PriorityQueue();
  Queue pending = new Queue();
  // Save mgr reference and register with mgr
  init(Policy M) {
    ourMgr = M;
    M <- registerReceiver(self);
  }
  // Intercept messages intended for base
  method rcv(Msg msg) {
    If request then store in sequenced
    Otherwise store in pending
  }
  // Determine next msg to deliver to base
  method ready() {
    If sequenced message seqMsg available then:
      deliver(seqMsg); seqNum++;
    Otherwise, deliver any pending messages
  }
  // Schedule sequenced deliver of request copy
  method copy(HttpRequest req, int seq) {
    Change "sender" of request to self
    Add request to sequenced
  }
}
```

Figure 3.11: `SequenceRequests` **Policy:** The `Sequencer` policy actor (top right) captures and sequences new requests. The `Receiver` policy actor (bottom right) ensures that requests are processed in the order of their sequencing. A separate `BcastActor` forwards copies of sequenced requests (not shown).



```
module client {
  accept proto...
     .
     .
  }
}
```

Figure 3.12: **DCL Mapping:** DCL specifications are mapped onto group manager actors.

3.6 Implementation

We implement DCL abstractions by mapping them to actor groups as defined in Section 2.3. In particular, a unique *group manager* actor encapsulates the semantics of the DCL abstraction associated with the group (see Figure 3.12). The group manager actor defines methods corresponding to the *init*, *accept*, *install*, and *method* rules of the corresponding specification. These methods are invoked when the corresponding rules are invoked in the specification.

The implementation provides a *group support API* (see below) which allows managers to perform various group operations. For example, an "admit" method requires the admission of a new actor to a group. To support this behavior, the group support API provides methods for changing actor membership privileges. Similarly, the `connect` and `install` commands are mapped to protocols within the API. Orthogonal to the API itself, the group support implementation is also responsible for ensuring that group membership privileges are enforced (two actors may only interact if they are a member of a common group).

3.6.1 Actor Implementation

The *Actor Foundry* [46] (henceforth, the Foundry) is a Java-based programming environment for developing actor programs. Specifically, the Foundry is structured as a collection of Java class libraries in which actors are modeled as Java objects. Java was used as the implementation language due to its embedded support for concurrency, networked-based applications, and heterogeneous execution environments. While there is an inherent performance degradation associated with Java implementations (due to byte-code interpretation), the Foundry is struc-

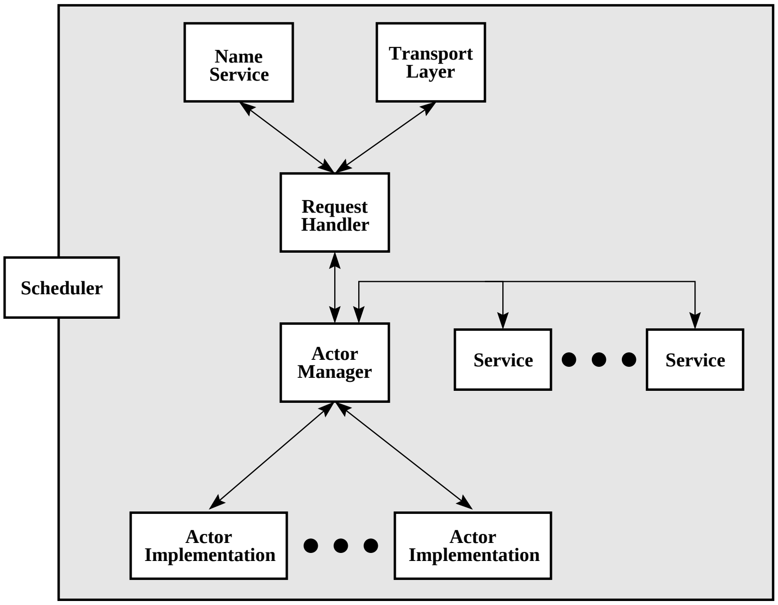


Figure 3.13: **Foundry Architecture:** The *ActorManager* provides basic actor functionality for all actors on a Foundry node. The *RequestHandler*, *NameService*, and *TransportLayer* are used to facilitate off-node services.

tured so that critical elements may be replaced with more efficient C-based implementations. The performance results presented in this section are based on a combined Java/C Foundry implementation.

An instance of the Foundry consists of a collection of *nodes*, each of which resides on a separate machine. A Foundry node is composed of a hierarchy of functional units for supporting various aspects of actor computation (see Figure 3.13). The *Actor Manager* manages all the actors on the local node. In particular, the manager provides access to message passing and actor creation services. A *Service* is a module which extends the basic functionality of the manager. That is, rather than require monolithic managers which encapsulate all possible functionality, services may be used as a modular way to extend the capabilities of the manager. The *Request Handler* facilitates node-to-node network communication by utilizing the *Name Service* and *Transport Layer.* The name service provides support for a globally unique naming system, while the transport layer facilitates low-level network communication. Finally, the

```
package osl.examples.helloworld;
import osl.manager.*;
public class HelloActor extends Actor {
   public void hello() throws RemoteCodeException {
      ActorName other = null;
      call(stdout, "print", "Hello ");
      other = create(WorldActor.class);
      send(other, "world");
   }
}
```

```
package osl.examples.helloworld;
import osl.manager.*;
public class WorldActor extends Actor {
   public void world() {
      send(stdout, "println", "World!");
   }
}
```

Figure 3.14: **“Hello World!” in the Foundry:** The `hello` method in `HelloActor` prints the string “Hello “, creates an instance of `WorldActor`, invokes the `world` method. The `world` method in `WorldActor` prints the string “World!”. A `call` is a method invocation with RPC semantics. A `send` is an asynchronous method invocation.

Scheduler is responsible for scheduling all threads associated with a node (including actors)[7]. Note that the *Actor Manager* abstracts over the node distribution of the Foundry. That is, actor programs may be developed independent of a specific configuration of the Foundry.

Each actor is an instance of the *Actor* class: the state and methods of the actor are the state and methods of an instance of this class (see Figure 3.14). Such objects are “active” in the sense that a private thread of control is used to invoke local methods. The *Actor* class provides primitives for message passing and dynamic creation of new actors. Applications are developed by extending the *Actor* class to define application actors.

Message passing is modeled as asynchronous method invocation. In addition, the Foundry provides support for actor migration, exception handling, and synchronous forms of communication such as remote procedure call. Each actor is managed by an instance of the *Actor Implementation* class, which provides a local mail queue and schedules the invocation of local methods in response to messages. Moreover, the *Actor Implementation* class defines the basic

[7] Originally, the inclusion of a scheduling module was necessary because of a deficiency in some Java implementations. In particular, the Java specification does not require a fair thread scheduler. As a result, many Java implementations include schedulers which are patently unfair, making actor programming difficult. After several iterations, we were able to develop a separate, fair scheduling module which performs nearly as fast as the default Java schedulers (less than 1% overhead). However, while more recent Java implementations support fair scheduling, we have retained the scheduling module as it provides a mechanism for experimenting with other scheduling paradigms (*e.g.* real-time).

Operation	Local	Remote
Message Passing	360 μsec/msg	13.4 ms/msg
Actor Creation	26.5 ms/create	50.5 ms/create
Scheduling	$< 1\%$	N/A

Figure 3.15: **Foundry Performance:** Message passing was sampled over 10000 iterations. Actor creation was sampled over 5000 iterations. Scheduling results were sampled over 10 iterations of *Fibonacci*(15).

implementation of actor services, such as message passing. Thus, the semantics of low-level actor behavior may be modified without changing existing actor code: a customized implementation of *Actor Implementation* may be used to provide the new behavior.

Figure 3.15 summarizes the performance of key aspects of the Foundry. The timing results were obtained from two Foundry nodes, each running on Sparc Ultra-2 workstations connected by 10baseT ethernet. The message passing results were obtained by timing an asynchronous send of 10000 messages both locally, and between local and remote nodes. Similarly, the actor creation results were obtained by timing the creation of 5000 actors locally and remotely. The scheduling results were obtained using an actor implementation of the Fibonacci function. Fibonacci is a useful indicator of scheduling efficiency because of the large number of actors created and the large volume of messages sent[8]. The scheduling results indicate the overhead of computing *Fibonnacci*(15) using the Foundry scheduler versus the default Java scheduler. Ten iterations were used to compute the resulting overhead.

As a distributed object toolkit, the Foundry compares favorably with other Java-based toolkits. For example, Javasoft's *Remote Method Invocation* toolkit [32] experiences a latency of approximately 5 ms for both local and remote interactions. By way of contrast, the RPC implementation in the foundry experiences latencies of 1.3 ms in the local case, and 28.7 ms in the remote case. The local case is much faster because all communication is isolated within a single Foundry node, whereas RMI makes an external call to the network libraries. In the remote

[8] Specifically, these two factors force a large number of context switches within the scheduler.

case, the Foundry is slower because of its heavily layered architecture. Whereas RMI makes a direct call to the network, the Foundry must route messages through the request handler, name service and transport layer. The result is a classic modularity/performance trade-off: the performance of the Foundry may be increased significantly by collapsing the architecture into a single network layer; however, doing so complicates the modular replacement of elements of the architecture in order to experiment with different implementations. As the Foundry is intended to be a research tool, we choose modularity in favor of performance.

A similar performance trade-off exists with respect to actor semantics. Specifically, actor semantics states that actors may communicate only by way of message passing. A practical consequence is that actors can not share state: such state may be used as a hidden channel for communication. As a result, all message arguments must be copied when delivering messages between actors on a local node. If actor semantics were relaxed (*e.g.* allowing shared state), then the performance of local message passing would be increased twofold. This results in local message passing performance which is comparable to a pure C implementation.

3.6.2 Meta-Architecture

To support meta-level customization, we have implemented a *meta-architecture API* which extends the basic actor APIs in the Foundry. The meta-architecture API is implemented by the class *MetaActorImpl* which is a subclass of *Actor Implementation*. Thus, a meta-actor is an instance of the *Actor* class which is managed by an instance of *MetaActorImpl*.

In addition to supporting the default actor methods (*e.g.* send, create), the *MetaActorImpl* class provides methods for creating and installing new meta-actors. Moreover, *MetaActorImpl* provides a default method behavior for each signal defined in the model. Currently, the implementation captures the most basic instantiation of the model. Thus, *MetaActorImpl* defines default behaviors for the `transmit`, `create` and `ready` signals. A meta-actor may provide custom behavior by overriding one or more of these methods. Similarly, *MetaActorImpl* provides methods for generating the basic notifications: `continue`, `newActor`, and `deliver`. These methods may be invoked in response to base-actor signals. Note, however, that notification

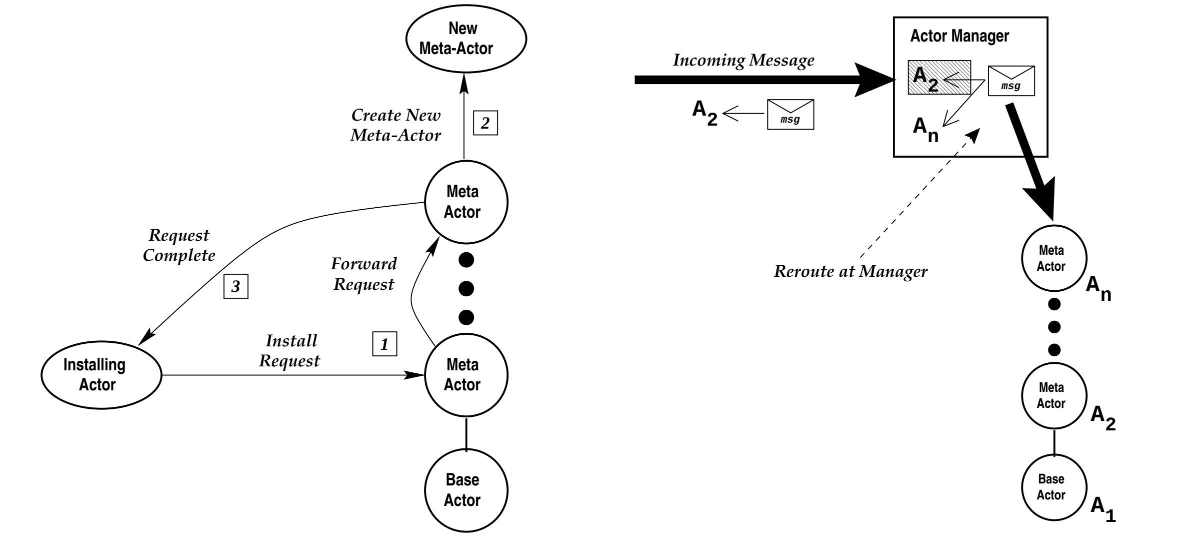


Figure 3.16: **Meta-Actor Implementation:** Left: New meta-actors are installed in a three step process. Right: Messages can be rerouted at the manager because all actors in a meta-level stack must reside on the same node.

generating methods are fixed ("final", in Java terms) and may not be customized. Similarly, the installation process is fixed by the implementation of *MetaActorImpl*.

Meta-actors are installed in a three step process (see Figure 3.16). In the first step, the installing actor sends a request to the target base actor and blocks until the installation completes. If the target actor already has a meta-actor installed, then the request is forwarded to the top actor in the meta-level stack. Thus, any actor in a meta-level may be used as a target for customization. The top actor in the stack creates the new meta-actor[9] and informs the requesting actor that the installation has completed. The actor which creates the new meta-actor will only perform the creation in between the processing of messages. This ensures that the meta-level stack is in a consistent state when the new customization is installed. Furthermore, we constrain the creation of new meta-actors so that they reside within the same Foundry node as their base actor. Note that this implies that all actors within a meta-level stack will reside on the same node. This constraint yields a more efficient implementation due to the large number of interactions between base and meta-actors. Once the installation completes, subsequent signals are automatically forwarded to the new meta-actor.

[9] Recall from Section 2.2 that actor creation is the only mechanism for installing meta-level customizations.

Although meta-level message redirection could also be handled within *MetaActorImpl*, we use a customized implementation of *Actor Manager* to provide a more efficient solution. In particular, because all actors within a meta-level stack are located within the same node, we may make stack routing decisions the moment a new message arrives at the manager[10]. This results in a significant performance improvement over a trivial implementation where messages are rerouted within the stack itself.

3.6.3 Group Support

The *group support API* is an extension of the *meta-architecture API* which provides support for group operations and encapsulation. Moreover, the group support API customizes certain aspects of the meta-architecture. For example, the group support API enforces the constraint that base and meta-actors must reside in different initial groups. We encapsulate group operations within the *GroupActorImpl* class, which extends *MetaActorImpl*. Note that the operations provided by the group support API are only accessible to group managers.

Group encapsulation is implemented by assigning *capabilities* [57] to each actor in the system. Capabilities are putatively unforgeable "access keys" used to verify that an actor has sufficient privilege to execute a particular operation. In our implementation, capabilities are formed from a *source key* consisting of a group part and a member part. The group part of the source key is the same for each actor in a group, while the member part is unique to each actor. The group part is unique to each group and is "secret". The member part of the source key may be determined from an actor's address. An encryption function is used to build a capability from the two parts of the source key. Note that when an actor creates a new actor, it passes the group part of its initial group to the new actor. This allows the new actor to generate a capability for its initial group (and hence send messages to other actors in the same group).

We use capabilities to allow actors within a common group to communicate with one another. In particular, each message sent by an actor is tagged with all the capabilities possessed by the

[10] This isn't quite true since an installation may be taking place at the time a new message arrives at a node. Thus, it may still be necessary to handle message routing within the meta-level stack itself. However, this is a rare occurrence in practice.

actor. When the message is received, a *capability check* is performed to determine whether or not the sender had sufficient privilege to send the message. The capability check proceeds as follows:

1. Let a_s be the member part of a source key determined from the sender's address. Let m be the message received. For each group, g, which the receiver is a member of:
   (a) Build a capability, c, using a_s and the group part of g.
   (b) If m contains a tag which is equal to c, then admit the message.
2. If no matching capability was found then reject the message.

If the check succeeds, then the message is delivered, otherwise it is rejected and sent back to the sender. Note that a capability check is not performed if the receiver is a group manager.

The group support API defines three operations for supporting groups: group creation, admission, and policy installation. Group creation allows a manager to create a new external group. A unique group part is created in order to build capabilities for the new group. The creating manager is passed the address of the new group manager once the creation completes. The admission operation admits a new actor by sending the secret group part of the source key of the manager's group. Once an actor obtains the group part of a source key, it may generate a capability which allows it to communicate with any other actor in the group. Note that capabilities are internal to the *GroupActorImpl* class. Thus, there is no danger of group secrets leaking to underlying user actors and being propagated to arbitrary locations.

Policy installation is a more complicated process involving a *creation tree* traversal (see Figure 3.17). The *GroupActorImpl* associated with each group member maintains a record of all the actors created by the member. Because the group manager is the first actor created in any group, we may build a tree of all the actors created in a group starting from the manager. We call this tree a *creation tree*. The creation tree for a DCL module consists of the creation tree rooted at the group manager. For a protocol, we associate a separate creation tree for each role. The creation tree for a role consists of the group manager and all the subsequent sub-trees

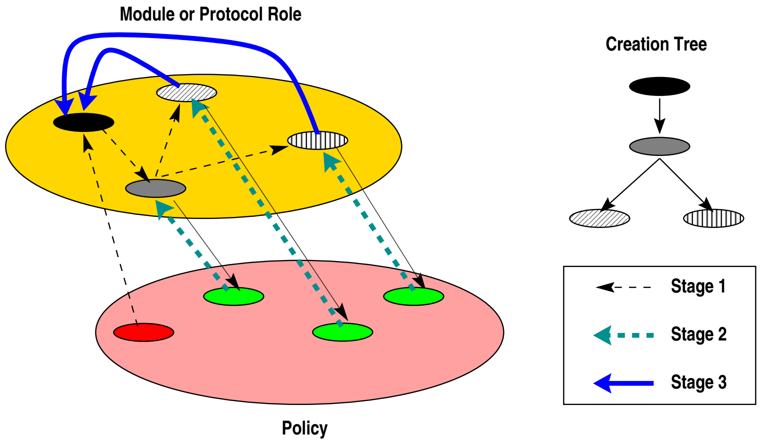


Figure 3.17: **Policy Installation:** Policies are installed in three stages: (1) the installation request is received and propagates down the group creation tree; (2) a meta-actor is installed on each member of the group; and (3) an acknowledgment is generated at the leaves of the creation tree to indicate the completion of the installation.

formed by the members of the role. In this case, the manager not only records the address of the actors it creates, but also the roles they are associated with.

A policy installation proceeds in three stages over the creation tree of the target. In the first stage, the requesting group sends an install request together with the secret group part of its source key. The installation request is then distributed by traversing the creation tree. In the second stage, a new meta-actor is installed on each actor in the creation tree (except for the manager). Note that this operation involves creating an actor in another group. However, because the secret group key was passed as part of the installation request, each new meta-actor may be properly added to the requester's group (*i.e.* it may create an appropriate capability for its initial group). In the third stage, the leaves of the creation tree send an acknowledgment to the manager indicating that the installation has completed.

Because the leaves of a creation tree may not be known to a group manager, it may be difficult for the manager to determine when the installation has completed. To circumvent this issue, we use a weighted reference counting algorithm similar to those used in some distributed garbage collection schemes (*cf.* [17] and [62]). The algorithm proceeds by tagging each installation request with a ratio $0 < \frac{p}{q} \leq 1$. The initial request sent by the manager is tagged with the value $1/n$ where n indicates the number of children below the manager in the creation tree. Each intermediate node tags the request with the value x/m where x is the value it received from its parent, and m is the number of children below it in the creation tree. When a leaf completes its installation, it sends the value it received back to the group manager. The group manager keeps a running total of all the values it has received. When this total reaches 1.0, the installation is complete.

A primitive implementation of the installation process described above would block at each stage of the installation. For our implementation, however, we are able to use a slightly more optimistic approach. In particular, we first associate a “version number” with each actor in a creation tree. The version number is constructed so that actors with structurally identical meta-level stacks will have identical version numbers. We then tag each installation request message with a logical time stamp and enforce the constraint that installation requests must be handled in tagged order. Similarly, we add the version number of each actor to every message it sends. We then enforce the constraint that an actor may only process a message if the version number in the message is less than the current version of the actor.

The combination of logical time stamps and version tagging allows the concurrent processing of installation requests. In particular, the addition of logical time stamps results in a First-In-First-Out (FIFO) multi-cast [25] over the actors in a creation tree. As a result, we can guarantee that each installation request is processed in the same order at each group actor. This eliminates the possibility of meta-actors being installed in the wrong order at different actors within a group. The use of version numbers in messages ensures that actor computation will be causally consistent with policy installation. That is, it will not be possible for an actor to process a message until it is in a state consistent with the state of the sender at the time the message was sent.

Operation	Overhead
Event Framework	8%/level
Installation	120%/level

Figure 3.18: **Meta-Level Performance:** Event framework overhead reflects the time required to pass events from base-actors to meta-actors. Installation overhead reflects the extra time required to install a meta-actor or a new actor.

3.7 Summary

In this chapter, we have provided an instantiation of the model described in Chapter 2. In particular, we have described a language for building architectures within the model, as well as an implementation which maps specifications into executable systems.

DCL provides a compositional approach for architectural development: rather than hard-code customization and resource management within protocols and modules, policies are used to modularly enforce such constraints. While this approach allows flexible system development, such mechanisms have limited use if they can not be mapped to efficient implementations. In particular, a key performance issue is the overhead associated with modular composition versus hard-coding customizations. In terms of DCL, this overhead is characterized by the cost of performing meta-level installation, and the cost of the event framework which translates base actor operations into events at meta-actors.

Figure 3.18 summarizes the performance of the event framework and meta-level installation. Event framework overhead was determined by timing the sending of 5000 messages between two actors. The first timing was performed without any meta-actors installed. The second test was performed with a meta-actor installed on each endpoint. The meta-actor installed on the sender redefined the `transmit` event and simply forwarded the message it received. The meta-actor installed on the receiver redefined the `rcv` method and simply delivered the message it received. The same test was performed for a meta-level stack of depths two and three on each endpoint. The result shows minimal overhead in the case of a meta-level stack of depth one. Overhead increases linearly with the depth of the stack.

As in the case of raw Foundry performance, the event framework overhead is significantly lower if we relax actor semantics and allow actors to share references to common state. In particular, with current actor semantics, the arguments of each event must be copied when moving an event from a base actor to its meta-actor. Because of the unique status of meta-actors, it could be argued that event state need not be copied. In this case, it is necessary to copy message arguments only when a message moves from one meta-level stack to another. In the current implementation, we provide a macro which allows the selection of copying or non-copying behavior.

Installation overhead was calculated by measuring the effect of a simple policy when installed on a *Fibonacci* module. We use the actor implementation of Fibonacci because of the large number of actors it creates. Recall that an appropriate meta-level stack is automatically installed each time a new actor is created within a module. Thus, Fibonacci is a useful indicator of installation overhead because of the number of times this installation must be performed. Overhead was computed by comparing the execution time of $Fibonacci(15)$ on an uncustomized instance of the module, to the execution time of a customized version of the module. The policy used to customize the module consisted of a simple meta-actor which intercepted `create` signals and simply forwarded them to the system.

As indicated in Figure 3.18, installation overhead is the major weakness of our approach. However, the overhead associated with installation is only a factor in applications in which actor creation is "bunched"; that is, in applications where many new actors are created rapidly. We argue that this is an implementation issue rather than a fault in the model. Specifically, the current implementation represents a meta-level stack literally in terms of a collection of separate actors. A more efficient approach would be to encapsulate meta-level state and behavior within a single actor[11]. This encapsulation would eliminate much of the extra synchronization necessary to create and install individual actors in a stack. Similarly, the performance of the event framework would be drastically improved as events would not have to be transmitted via message passing.

[11] Of course, other modifications would be necessary to allow the actor to be associated with multiple addresses, one for each meta-actor.

Chapter 4

Formal Semantics

In this chapter, we provide a formal semantics for the model described in Chapter 2. The semantics provides a basis for reasoning about composition. Specifically, we describe two types of composition:

- **Composition of Interactions:** This form of composition is characterized by message passing between overlapping groups: given two groups a and b, the composition is characterized by the messages which are sent from an actor with initial group a, and received by an actor with initial group b, and vice versa.
- **Composition of Customizations:** This form of composition is characterized by the signals and notifications sent between base and meta-actors.

Using a formal model for composition, we characterize "architectural compatibility". That is, we define the conditions under which DCL protocols may be used to interconnect a collection of DCL modules. Similarly, we define conditions under which a DCL policy will conflict with underlying behavior of modules or protocols it customizes.

We develop our semantics as an extension of Abstract Actor Structures (AAS) [55], a concurrent rewriting [40] formulation of actor semantics. While concurrent rewriting establishes the basic properties of the semantics, we use the notion of interaction semantics [56] to define composability properties. The semantics is developed in three stages.

In the first stage, we develop a semantics for local actor computation. As with AAS, we abstract over a particular linguistic representation of actor behavior: actors are represented abstractly as states and behavior types. In accordance with the model described in Chapter 2, we provide a fine-grained interpretation of actor behavior: actor computation is represented as a series of atomic event processing steps. In response to an event, an actor performs some local (*i.e.* non-visible) computation and generates a new event. This approach should be compared to the "super-steps" of AAS where the processing of a single message may result in several new actors and messages. In practice, our approach is nearly equivalent to an interpretation of AAS where certain messages may restrict further processing until a particular reply message is received (*i.e.* blocking on signals, waiting for notifications). Using the fine-grained approach, we introduce a basic set of rules that allows us to recover the traditional semantics for actors.

In the second stage of development, we extend traditional actor semantics to support meta-level customization. To accomplish this, we derive a new event-processing mechanism and introduce rules that allow actors to process events generated by other actors. This approach allows a meta-actor to intercept the service requests made by a base actor. Once the meta-level semantics has been defined, we add support for group operations required by our model. Note that group operations imply additional constraints on meta-level semantics. However, we derive meta-level semantics separately to illustrate their utility independent of the actor group model.

Finally, in the third stage of development we define an interaction semantics based on interactions between actor groups. Using this semantics, we establish compatibility conditions that define the composability of groups. This semantics provides a definition of composability based on group-to-group interactions.

The following notation is used in this chapter. The expression $f : D \rightarrow R$ represents a total function f with domain D and range R. The expression $f : D \overset{\circ}{\rightarrow} R$ represents a partial function with the same domain and range. The expression $\mathbf{P}_n[S]$ indicates the set of all subsets s of S with $|s| \leq n$. The expression $\mathbf{P}_\omega[S]$ indicates the set of all finite subsets of S.

4.1 Preliminaries

Concurrent rewriting is a formal model that allows one to reason about concurrent systems in terms of state transitions. A key strength of concurrent rewriting is its ability to represent many models of concurrency within a common framework. In addition to actors, for example, other models such as CCS [42] and π-calculus [43, 44] may be captured within the concurrent rewriting framework (see [41]). Although we make no such comparisons here, the generality of concurrent rewriting may allow the translation of important techniques between different formalisms. In particular, it may be seen that the notion of compatibility defined in Wright [10] (based on CCS) is equivalent to our notion of interaction compatibility. We provide a brief summary of concurrent rewriting below. A more complete description may be found in [40].

A rewriting logic theory is composed of a *signature*, (Σ, E), together with rewrite theory rules. The signature defines the algebraic portion of a theory. In particular, the set Σ defines the function symbols of the theory, while the set E consists of Σ-equations. A Σ-algebra is a set V where each $f \in \Sigma$ of n arguments is associated with a function $f : V^n \to V$. The symbol T_Σ denotes the Σ-algebra of ground Σ-terms (*i.e.* 0-ary functions in Σ). Similarly, the symbol $T_{\Sigma,E}$ denotes the Σ-algebra of equivalence classes of ground Σ-terms modulo (*i.e.* equivalent with respect to) the equations E. It is often convenient to use variables as place holders in rewriting expressions. For such expressions, if X denotes a countable set of variables, then the symbols $T_\Sigma(X)$ and $T_{\Sigma,E}(X)$ denote, respectively, the Σ-algebra of Σ-terms with variables in X, and the Σ-algebra of equivalence classes of Σ-terms with variables in X modulo the equations E. Similarly, given a term $t \in T_\Sigma(\{x_1, ..., x_n\})$ and terms $u_1, ..., u_n$, we write $t(u_1/x_1, ..., u_n/x_n)$ to denote the term obtained from t by simultaneously substituting each u_i for each x_i. Finally, for a term t, the symbol $[t]$ denotes the E-equivalence class of t.

Given a signature, (Σ, E), the rewrite rules of a theory are represented by a pair (L, R) where L is a set of *labels*, and R is a set of 3-tuples with $R \subset L \times T_{\Sigma,E}(X) \times T_{\Sigma,E}(X)$. A rule instance, $(r, [t], [t'])$, is interpreted as a *labeled sequent* and denoted as $r : [t] \longrightarrow [t']$. Sequents define the *sentences* of rewriting logic. Moreover, the "interesting" derivations in a particular theory will include one or mode applications of labeled sequents.

A *rewrite theory*, then, consists of a 4-tuple $\mathbb{R} = (\Sigma, E, L, R)$. The theory $\mathbb{R}$ is said to *entail* the sequent $[t] \longrightarrow [t']$ if and only if $[t] \longrightarrow [t']$ may be inferred from the finite application of the following *rules of deduction.* The rules of deduction establish a rewrite theory as a logic and are defined as follows:

- **Reflexivity:** For each $[t] \in T_{\Sigma,E}(X)$,

$$\frac{}{[t] \longrightarrow [t]}$$

- **Congruence:** For each $f \in \Sigma$ of $n \in \mathbb{N}$ arguments,

$$\frac{[t_1] \longrightarrow [t'_1] \ \ ... \ \ [t_n] \longrightarrow [t'_n]}{[f(t_1, ..., t_n)] \longrightarrow [f(t'_1, ..., t'_n)]}$$

- **Replacement:** For each rule $r : [t(x_1, ..., x_n)] \longrightarrow [t'(x_1, ..., x_n)]$ in R,

$$\frac{[w_1] \longrightarrow [w'_1] \ \ ... \ \ [w_n] \longrightarrow [w'_n]}{[t(\bar{w}/\bar{x})] \longrightarrow [t'(\bar{w'}/\bar{x})]}$$

- **Transitivity:**

$$\frac{[t_1] \longrightarrow [t_2] \ \ [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}$$

In less formal terms, a rewriting logic may be characterized by a set of *sorts*, a set of *operations*, a set of *equational rules*, and a set of *rewrite rules.* Sorts, operations and equational rules establish the algebraic portion of the logic. In particular, sorts are used to categorize terms in the logic, while operations are used to build well-formed terms. Equational rules define equivalence classes of terms in the logic. Similarly, rewrite rules define relations for matching and replacing terms.

In [55], the traditional operational semantics of actors (*cf.* [5]) has been reformulated in terms of rewriting logic. Actor configurations in this theory are represented by elements of the following sorts:

- $\mathcal{A}$ - The sort of actor *names.*
- $\mathcal{V}$ - The sort of *values.*
- $\mathcal{S}$ - The sort of actor *states.*
- $\mathcal{F}$ - The sort of actor *fragments.*

- $\mathcal{C}$ - The sort of *configurations.*

A typical term in this semantics is represented as:

$$\langle (s_1)_{a_1}, ..., (s_n)_{a_n}, a_i \lhd v_i, ..., a_j \lhd v_j \rangle$$

where $(s_i)_{a_i}$ specifies an actor in state s_i with address a_i, and $a_j \lhd v_j$ specifies a message with destination a_j and contents v_j. Unlike the operational semantics, the rewriting formulation abstracts over the individual behavior of actors. Instead, actor behavior is captured abstractly in terms of a delivery function, $Deliv$, and execution functions, Ex and $\#new$. Similarly, properties of a configuration such as "enabled sets" (*i.e.* those actors enabled for delivery or execution) are captured by an "enabled for delivery" relation, En_d, and an "enabled for execution" set, En_{ex}. Finally, the name space of actor terms may be controlled by a restriction operator, $\lceil a_p, ..., a_q$, which, if applied to a fragment, restricts the external visibility of internal names to the set $a_p, ..., a_q$.

The resulting semantics is greatly simplified by abstracting away low-level actor behavior. In particular, actor semantics may be characterized by just four rewrite rules[1]:

- **Execute:** $\mathtt{e}(a, s) : (s)_a \Rightarrow Ex((s)_a)(\vec{a})\lceil\{a\}$
  if $En_{ex}(a, s)$ and $\vec{a}$ is a list of $\#new(a, s)$ distinct actor names disjoint from a and $acq(s)$.
- **Deliver:** $\mathtt{d}(a, s, v) : \{(s)_a, a \lhd v\} \Rightarrow Deliv(a, s, v)$
  if $En_d(a, s, v)$
- **In:** $\mathtt{i}(F, a \lhd v) : \langle F \rangle \Rightarrow \langle \{F, a \lhd v\} \rangle$
  if $a \in recep(F)$
- **Out:** $\mathtt{o}(F, R, a \lhd v) : \langle \{F, a \lhd v\} \lceil R \rangle \Rightarrow \langle F \lceil R \cup (acq(v) \cap recep(F)) \rangle$
  if $a \notin recep(F)$

where **Execute** defines an execution step, **Deliver** defines a message delivery step, and **In** and **Out** define interactions with the external environment. The expression $Ex((s)_a)(\vec{a})\lceil\{a\}$ defines the fragment resulting from an actor computation step. In general, this fragment has the form:

$$(s')_a, (s_1)_{a_1}, ..., (s_n)_{a_n}, a_i \lhd v_i, ..., a_j \lhd v_j \lceil \{a\}$$

[1] Technically, we also require several hygiene conditions. These are normally expressed as axioms over particular relations (including Ex and $Deliv$), and constitute the equational rules for the semantics.

where actor a is in the new state s', the actors with addresses a_1 through a_n represent new actors (thus the restriction that a and $acq(s)$ be disjoint from the set $\{a_1, ..., a_n\}$), and $a_i \lhd v_i$ through $a_j \lhd v_j$ represent messages sent by a during the computation step. Similarly, the expression $Deliv(a, s, v)$ in the deliver rule results in a new fragment of the form $(s')_a$ reflecting the fact that an actor's state may change while consuming a message. Finally, the "in" and "out" rules provide a mechanism for interacting with the external environment. This feature has long been present in actor semantics and reflects the view that actor configurations are open systems in which interaction patterns may not be determined solely by the behavior of internal actors.

The rewriting formulation of actor semantics is more abstract than the traditional operational semantics. In particular, rewrite rules define actor computation in terms of "super steps". That is, a single execution step may result in several new actors and several new messages. By way of contrast, the operational semantics is a fine-grained representation of actors: each individual actor operation is reflected in the semantics, from individual lambda term evaluation, through actor creation and message passing. However, we may carry this level of abstraction one step further and abstract over internal actor computation altogether. This is the approach taken in *interaction semantics.*

Talcott has developed an interaction semantics for actors in [56]. This semantics is similar to the rewriting formulation of actors, except that only three transitions are defined:

- **Silent:** $\langle\,\rangle : C \Longrightarrow C'$ if $\tau : C \Longrightarrow C'$
  where $\tau : C \Longrightarrow C'$ represents an *internal transition* (*i.e.* actor execution or message delivery).
- **In:** $\texttt{in}(a \lhd v) : C \Longrightarrow C, a \lhd v$ if $a \in recep(C)$.
- **Out:** $\texttt{out}(a \lhd v) : C, a \lhd v \lceil R \Longrightarrow C \lceil R \cup X$
  if $a \notin recep(C)$, and $X = acq(v) - (recep(C) \cup extern(C))$.

Because internal transitions are captured solely by the silent transition, the interactions of interest in this model are those with the external environment. This view of actor computation has been used to derive correspondence conditions between actor configurations [52]. For example, a configuration that represents an implementation may be shown to be a refinement of a configuration which represents a specification.

Our goal in developing a rewriting semantics with an interaction component is to leverage the flexibility of interaction semantics in order to show that one configuration is "interaction compatible" with another configuration. For example, we wish to show that the interaction semantics of configurations representing modules are preserved under composition with configurations representing protocols. In our case, however, the boundary for interactions is represented by groups rather than configurations.

4.2 A Rewriting Semantics for Actors

To develop a semantics for actors we first start with an abstract description of actor computation. In particular, we model actor computations as an alternating sequence of internal (*i.e.* non-visible) computation steps followed by external (*i.e.* visible) computation steps. Intuitively, this model corresponds to an operating system view of an actor: an actor performs some internal computation defined by its behavior, but may periodically request a service (*e.g.* message passing, new actor creation) which only the operating system can provide. Service requests are modeled by *events.* An actor is blocked (*i.e.* unable to process other events) until its request is handled. This corresponds to the interpretation that an actor can not use a resource until that resource has been granted or denied. An event generated by an actor is called a *signal*, while an event received by an actor is called a *notification.* Thus, a signal indicates a request for service, whereas a notification indicates that a request has been handled in a particular fashion. Each signal is associated with one or more notifications that indicate the outcome of a request.

In deriving a rewriting semantics for actors, we follow the conventions used in [55]. In particular, terms derived from our equational axioms and rewrite rules are considered to be well-formed only if certain constraints are met. For example, to introduce an operation f such that $f(x_1, ..., x_n)$ is well-formed and of sort Y just if each x_i is of sort X_i, and the condition $\phi(x_1, ..., x_n)$ holds, we write:

$$f(\_, ..., \_) : X_1 \times ... \times X_n \longrightarrow Y$$

$$f(x_1, ..., x_n) : Y \text{ if } \phi(x_1, ..., x_n)$$

Moreover, we adopt the convention that only "fair" rewrite derivations are considered in the semantics. The construction of such derivations captures the usual notion of fairness in an actor system (*i.e.* observational fairness) and is rather complex to define. We do not derive fair rewrites here. Rather, we refer the reader to the construction given in [55]. Note that the rewrite systems defined in this thesis are somewhat more restrictive than the usual actor semantics. That is, the "interesting" paths in our semantics are a subset of the fair paths in a traditional actor system. Thus, our semantics causes no inconsistencies with the construction of fair paths provided in [55].

The following sorts are used in the semantics:

- $\mathbb{V}$ - Communicable values
- $\mathbb{A}$ - Addresses with $\varsigma \in \mathbb{A}$ and $\mathbb{A} \subset \mathbb{V}$
- $\mathbb{S}$ - States
- $\mathbb{B}$ - Behavior types with $\mathbb{B} \subset \mathbb{V}$
- $\mathbb{E}$ - Finite set of events with $\perp \in \mathbb{E}$
- $\mathbb{P}$ - Processing states with $\mathbb{P} = \{?, !\}$
- $\mathbb{F}$ - Fragments
- $\mathbb{C}$ - Configurations

The sort $\mathbb{V}$ represents the set of all values that may be communicated among actors. $\mathbb{A}$ is a countable set which we use for naming actors. The special name ς represents the address of the "system" actor and is always included in the set of names for an actor system. Note that actor addresses are communicable values. Elements of $\mathbb{B}$ are used as a convenience to type actor behaviors, while elements of $\mathbb{S}$ are used to represent actor states. Note that elements of $\mathbb{B}$ are also communicable values. The sort $\mathbb{E}$ represents the set of all actor events. The special symbol $\perp$ is defined as the "null" event and is always included in the set of events for an actor system. The sort $\mathbb{P}$ defines the two processing states of an actor. The symbol ? denotes a "ready" actor, while the symbol ! denotes a "running" actor. Finally, instances of sorts $\mathbb{F}$ and $\mathbb{C}$ represent actor fragments and actor configurations, respectively.

Actors are modeled as 6-tuples, (p, a, b, s, l, t), with attributes:

$\_(\_,\ \_\,|\,\sigma : [\_]\ \lambda : [\_]\ \tau : [\_]) : (\mathbb{P} \times \mathbb{A} \times \mathbb{B} \times \mathbb{S} \times \mathbb{E} \times \mathbb{A}) \xrightarrow{\circ} \mathbb{F}$

Denotes an actor. The actor $p\,(a\,,\,b\,|\,\sigma : [s]\ \lambda : [l]\ \tau : [t])$ is in processing state p, with address a, behavior b, state s, last event generated l, and transition map t. If $p = ?$, then the actor is considered to be "ready", otherwise the actor is "running". By convention, the vertical separator, $|$, will always be used to separate the fixed attributes (on the left) from the run-time attributes (on the right) of an actor.

$\_ \blacktriangleleft (\_,\_,\_) : (\mathbb{A} \times \mathbb{A} \times \mathbb{E} \times \mathbb{V}) \longrightarrow \mathbb{F}$

Denotes an event message. In the event message $r \blacktriangleleft (s, e, v)$ the address r is the receiver, s is the sender, e is the event, and v is the contents.

$\diamond : \mathbb{F}$

Denotes the empty fragment.

$\_\,,\,\_ : (\mathbb{F} \times \mathbb{F}) \xrightarrow{\circ} \mathbb{F}$

Denotes fragment composition which is associative, commutative and has identity $\diamond$. Moreover, $F_1\,,\,F_2 \in \mathbb{F}$ if $\mathit{recep}(F_1) \cap \mathit{recep}(F_2) = \emptyset$.

$[\_]\lceil\{\_\} : \mathbb{F} \times \mathbf{P}_\omega[\mathbb{A}] \xrightarrow{\circ} \mathbb{F}$

Denotes a restriction of a fragment. Moreover, $[F]\lceil\{a_1, ..., a_n\} \in \mathbb{F}$ if $\{a_1, ..., a_n\} \subset \mathit{recep}(F)$.

$\langle\_\rangle : \mathbb{F} \xrightarrow{\circ} \mathbb{C}$

Denotes a configuration.

Figure 4.1: **Term Constructors:** Operations used to build terms in the rewriting semantics.

- $p \in \mathbb{P}$ – A processing state.
- $a \in \mathbb{A} \setminus \{\varsigma\}$ – An address.
- $b \in \mathbb{B}$ – A behavior.
- $s \in \mathbb{S}$ – A state.
- $l \in \mathbb{E}$ – The last event generated by an actor.
- $t \in \mathbb{A}$ – The transition map for an actor.

The behavior and address of an actor are fixed when it is created. The state, last event generated, processing state, and transition map are run-time attributes determined by the events processed by the actor. For example, the transition map for an actor indicates the name of the actor which will process its signals. The value of this attribute may be changed at run-

$\mathit{block} \subset ((\mathbb{E} \setminus \{\bot\}) \times (\mathbb{E} \setminus \{\bot\}))$

Represents the blocking relation. If $(e_1, e_2) \in \mathit{block}$, then an actor generating event e_1 may only be resumed by an event of type e_2.

$\mathit{nextEvent} : (\mathbb{S} \times \mathbb{B}) \overset{\circ}{\longrightarrow} (\mathbb{E} \times \mathbb{V})$

Gives the next event and value generated by a running actor with a particular state and behavior.

$\mathit{nextState} : (\mathbb{S} \times \mathbb{B} \times \mathbb{E} \times \mathbb{V}) \overset{\circ}{\longrightarrow} \mathbb{S}$

Gives the state of an actor after receiving an event.

$\mathit{newInstance} : (\mathbb{B} \times \mathbb{V}) \overset{\circ}{\longrightarrow} \mathbb{S}$

Gives the state of a new actor created with a particular behavior and "initial" argument.

$\mathit{recep} : \mathbb{F} \longrightarrow \mathbf{P}_\omega[\mathbb{A}]$

Gives the set of receptionists contained in a fragment.

$\mathit{acq} : (\mathbb{S} \cup \mathbb{V}) \longrightarrow \mathbf{P}_\omega[\mathbb{A}]$

Gives the set of acquaintances contained in a state or value.

$\mathit{extern} : \mathbb{F} \longrightarrow \mathbf{P}_\omega[\mathbb{A}]$

Gives the set of external actors referenced in a fragment.

$\widehat{\_} : \left(\mathbb{A} \xrightarrow[1-1]{} \mathbb{A}\right) \longrightarrow \left(\mathbb{S} \cup \mathbb{V} \cup \mathbb{F} \xrightarrow[1-1]{} \mathbb{S} \cup \mathbb{V} \cup \mathbb{F}\right)$

Lifts a renaming function (*i.e.* a bijection $\rho : \mathbb{A} \longrightarrow \mathbb{A}$) to states, values and fragments such that ρ and $\widehat{\rho}$ agree on actor names.

Figure 4.2: **Relations:** Relations used to manipulate actor configurations.

time if the actor is customized by a meta-actor. A configuration of actors is constructed using the operations and relations given in Figures 4.1 and 4.2.

The *recep*, *extern* and *acq* relations are used to define several important hygiene conditions over actor configurations, and establish the equational portion of the semantics. *recep* gives the set of receptionists for an actor fragment and is defined inductively as follows[2]:

$$\begin{array}{rcl}
\mathit{recep}(p\,(\,a\,,\,b\,|\,\sigma : [s]\ \lambda : [l]\ \tau : [t])) & = & \{a\} \\
\mathit{recep}(r \blacktriangleleft (s, e, v)) & = & \emptyset \\
\mathit{recep}(\diamond) & = & \emptyset \\
\mathit{recep}(F_1, F_2) & = & \mathit{recep}(F_1) \cup \mathit{recep}(F_2) \\
\mathit{recep}([\,F\,]\,\lceil\{\,a_1, ..., a_n\,\}) & = & \{a_1, ..., a_n\}
\end{array}$$

[2] Henceforth, we assume that expressions such as $\mathit{recep}(F_1, F_2)$ are well defined only in the case that $F_1, F_2 \in \mathbb{F}$.

acq gives the set of acquaintances encoded within a state or value. Because we do not wish to fix a particular representation for states or values, we simply assume that *acq* is well defined on $\mathbb{S} \cup \mathbb{V}$ with the additional constraint that $acq(a) = \{a\}$ for all $a \in \mathbb{A}$.

extern gives the set of external actors referenced in a fragment and is defined inductively as follows:

$$\begin{array}{rcl}
extern(p\,(\,a\,,\,b\,|\,\sigma:[s]\;\lambda:[l]\;\tau:[t])) & = & (acq(s) \cup \{t\}) \setminus \{a\} \\
extern(r \blacktriangleleft (s,e,v)) & = & acq(v) \cup \{r,s\} \\
extern(\diamond) & = & \emptyset \\
extern(F_1,F_2) & = & (extern(F_1) \cup extern(F_2)) \setminus recep(F_1,F_2) \\
extern([\,F\,]\lceil\{\,a_1,...,a_n\,\}) & = & extern(F)
\end{array}$$

A key feature of actor configurations is that their semantics do not depend on a particular choice of actor addresses. This attribute is formalized by the notion of a renaming bijection, $\rho : \mathbb{A} \longrightarrow \mathbb{A}$, where we require $\rho(\varsigma) = \varsigma$. The relation $\widehat{\_}$ extends a renaming function to states, values and fragments according to the following axioms:

$$\begin{array}{rcl}
\widehat{\rho}(a) & = & \rho(a)\ \forall\ a \in \mathbb{A} \\
\widehat{\rho}(s) & \in & \mathbb{S}\ \ \forall\, s \in \mathbb{S} \\
\widehat{\rho}(v) & \in & \mathbb{V}\ \ \forall\, v \in \mathbb{V} \\
\widehat{\rho}(p\,(\,a\,,\,b\,|\,\sigma:[s]\;\lambda:[l]\;\tau:[t])) & = & p\,(\,\widehat{\rho}(a)\,,\,b\,|\,\sigma:[\widehat{\rho}(s)]\;\lambda:[l]\;\tau:[\widehat{\rho}(t)]) \\
\widehat{\rho}(r \blacktriangleleft (s,e,v)) & = & \widehat{\rho}(r) \blacktriangleleft (\widehat{\rho}(s),e,\widehat{\rho}(v)) \\
\widehat{\rho}(\diamond) & = & \diamond \\
\widehat{\rho}(F_1,F_2) & = & \widehat{\rho}(F_1),\widehat{\rho}(F_2) \\
\widehat{\rho}([\,F\,]\lceil\{\,a_1,...,a_n\,\}) & = & [\,\widehat{\rho}(F)\,]\lceil\{\,\widehat{\rho}(a_1),...,\widehat{\rho}(a_n)\,\} \\
\widehat{\rho_0 \circ \rho_1} & = & \widehat{\rho}_0 \circ \widehat{\rho}_1
\end{array}$$

Finally, several "restriction" axioms are necessary in order to allow fragments to be manipulated around the restriction operator:

$$\begin{array}{rcl}
F & = & [\,F\,]\lceil\{\,recep(F)\,\}\ \forall\ F \in \mathbb{F} \\
[\,[\,F_0\,]\lceil\{\,R_0\,\}\,,\,F_1\,]\lceil\{\,R\,\} & = & [\,F_0\,,\,F_1\,]\lceil\{\,R\,\}\text{ if }\ (recep(F_0) \setminus R_0) \cap extern(F_1) = \emptyset \\
[\,F\,]\lceil\{\,R\,\} & = & [\,\widehat{\rho}(F)\,]\lceil\{\,R\,\}\text{ if }\rho(x) = x\ \forall\ x \in R \cup extern(F)
\end{array}$$

More specific hygiene conditions are required for certain rewrite rules. We define these extraneous conditions as they arise.

4.2.1 Basic Computation Steps

Actor computation based on event processing is similar to a restricted form of the usual message-based semantics for actors. In particular, by fixing the sort of events, $\mathbb{E}$, and defining the relation *block*, actor computation over a particular configuration, $\langle F \rangle$, may be defined using two rules:

`[request]`

$$!\big(a\,,\,b\,|\,\sigma:[s]\ \lambda:[l]\ \tau:\big[a'\big]\big) \longrightarrow\ ?\big(a\,,\,b\,|\,\sigma:[s]\ \lambda:[e]\ \tau:\big[a'\big]\big)\,,\,a' \blacktriangleleft (a,e,v)$$

$$\text{if}\quad \mathit{nextEvent}(s,b) \ = (e,v)$$

`[compute]`

$$?\big(a\,,\,b\,|\,\sigma:[s]\ \lambda:[l]\ \tau:\big[a'\big]\big)\,,\,a \blacktriangleleft (a',e,v) \longrightarrow\ !\big(a\,,\,b\,|\,\sigma:\big[s'\big]\ \lambda:[l]\ \tau:\big[a'\big]\big)$$

$$\begin{array}{lrll} \text{if} & (l,e) & \in \mathit{block} & \text{and} \\ & \mathit{nextState}(s,b,e,v) & = s' & \end{array}$$

The `request` rule transforms a running actor into a ready actor and an event pair. The *nextEvent* function determines the next signal generated by an actor based on its behavior and current state. The transition map of an actor determines where the new signal should be processed. After generating a signal, an actor is blocked until an appropriate notification event is received.

The `compute` rule allows a blocked actor to become active by processing a notification event. The *nextState* function determines the new state of an actor after it has processed a notification. Thus, the `compute` rule abstracts over the internal computation performed by an actor upon receiving a particular notification. Note that the *block* relation defines the set of *notification* events which may be received and processed after an actor generates a particular *signal* event. Moreover, note that an actor will be "stuck" if it generates a signal e such that there does not exist $(x, y) \in \mathit{block}$ with $x = e$.

We impose two hygiene conditions over the *nextEvent* and *nextState* functions[3]:

$$\begin{array}{rcl} \mathit{nextEvent}(s,b) = (e,v) & \Longrightarrow & \mathit{acq}(v) \subset \mathit{acq}(s) \\ \mathit{nextState}(s,b,e,v) = s' & \Longrightarrow & \mathit{acq}(s') \subset \mathit{acq}(s) \cup \mathit{acq}(v) \end{array}$$

[3] These conditions correspond to the "execution axioms" defined in [55].

The first constraint states that an event generated by an actor may not contain an address that was not stored in the state of the actor when the event was generated. The second constraint states that the addresses stored in an actor's state are a function of its state history and the events it receives. Specifically, an actor may only accumulate addresses by receiving events.

It is interesting to note that, using the two rules above, we may define a simple, fixed configuration of actors. In this context, the sort of events, $\mathbb{E}$, is interpreted as the "types" of messages which may be exchanged between actors. The transition map for each actor fixes the communication topology of the configuration. Using event processing as a basis, it may be possible to derive other interesting models besides actors.

4.2.2 Traditional Actor Semantics

We recover the traditional semantics for actors by fixing the sort of events, $\mathbb{E}$, as:

$$\mathbb{E} = \{\mathbf{transmit}, \mathbf{create}, \mathbf{ready}, \mathbf{complete}, \mathbf{newActor}, \mathbf{deliver}, \perp\}$$

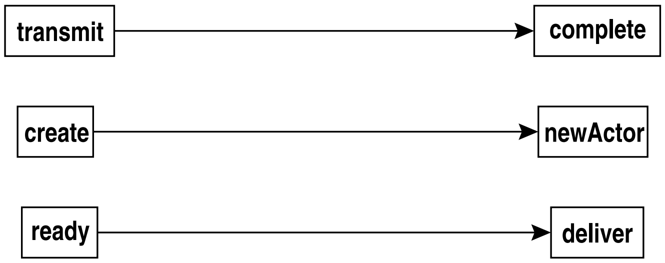


Figure 4.3: **Basic Actor Transitions:** The *block* relation for the traditional formulation of actor semantics.

The *block* relation is defined as shown in Figure 4.3. We also introduce the *message* operation, $\_ : \_ \lhd \_ : (\mathbb{A} \times \mathbb{A} \times \mathbb{V}) \to \mathbb{F}$, where $a : a' \lhd v$ represents a message sent with contents v to the actor with address a from the actor with address a'[4]. We add the following axiomatic constraints for actor messages:

[4] Strictly speaking, the basic actor semantics does not require that messages be annotated with the address of the sender. However, we require this information for our derivation of interaction semantics later in the chapter.

$$\begin{array}{rcl} \mathit{recep}(a : a' \lhd v) & = & \emptyset \\ \mathit{extern}(a : a' \lhd v) & = & \mathit{acq}(v) \cup \{a, a'\} \\ \widehat{\rho}(a : a' \lhd v) & = & \widehat{\rho}(a) : \widehat{\rho}(a') \lhd \widehat{\rho}(v) \end{array}$$

The normal operations provided by an actor configuration (*e.g.* message passing, actor creation) are defined by the following rules:

`[send]`

$$\varsigma \blacktriangleleft (a, \textbf{transmit}, \{a', v\}) \quad \longrightarrow \quad a \blacktriangleleft (\varsigma, \textbf{complete}, \{\})\, ,\, a' : a \lhd v$$

`[create]`

$$A\, ,\, \varsigma \blacktriangleleft (a, \textbf{create}, \{b', v\}) \quad \longrightarrow \quad [\, A\, ,\, a \blacktriangleleft (\varsigma, \textbf{newActor}, \{a'\})\, ,\, A'\, ,\, \varsigma \blacktriangleleft (a', \textbf{ready}, \{\})\,]\, \lceil \{\, a\, \}$$

$$\begin{array}{lrll} \text{if} & A & = ?\, (\, a\, ,\, b \,|\, \sigma : [s]\ \lambda : [\textbf{create}]\ \tau : [\varsigma]) & \text{and} \\ & A' & = ?\, (\, a'\, ,\, b' \,|\, \sigma : [s']\ \lambda : [\textbf{ready}]\ \tau : [\varsigma]) & \text{and} \\ & \mathit{newInstance}(b', v) & = s' & \text{and} \\ & a' & \notin \mathit{acq}(s) \cup \{a\} \cup \mathit{acq}(v) & \end{array}$$

`[ready]`

$$\varsigma \blacktriangleleft (a, \textbf{ready}, \{\})\, ,\, a : a' \lhd v \quad \longrightarrow \quad a \blacktriangleleft (\varsigma, \textbf{deliver}, \{v\})$$

`[in]`

$$\langle F \rangle \quad \longrightarrow \quad \langle F\, ,\, a : a' \lhd v \rangle$$

$$\text{if} \quad a \quad \in \mathit{recep}(F)$$

`[out]`

$$\langle [\, F\, ,\, a : a' \lhd v\,]\, \lceil \{\, R\, \} \rangle \quad \longrightarrow \quad \langle [\, F\,]\, \lceil \{\, R \cup (\mathit{acq}(v) \cap \mathit{recep}(F))\, \} \rangle$$

$$\text{if} \quad a \quad \notin \mathit{recep}(F)$$

Under this rule set, signals are modeled as requests handled by a "system actor" denoted with the address ς. Similarly, notifications are sent from ς to indicate that a particular request has been handled. The `send` rule transforms a **transmit** signal into a message and a **complete** notification indicating that the send has been processed. The `create` rule transforms a **create** signal into a new actor and a **newActor** notification. The restriction on the right side of the rule ensures that fragment composition respects actor name propagation: external actors may only learn of new actors by receiving messages. Note that the new actor has a fresh address.

Also, the transition map is updated so that the new actor may generate events. The `ready` rule consumes a **ready** signal and transforms an incoming message into a **deliver** notification. Finally, the `in` and `out` rules allow interactions with the external environment.

Under this semantics, we assume that actor signals may only be handled by ς. Moreover, the system actor may resume only those actors which block on a **transmit**, **create**, or **ready**. That is, an actor generating any other event will be "stuck". The `create` rewrite guarantees that these constraints are enforced over dynamically created actors (*i.e.* those created as a result of the `create` rewrite). To ensure that an *initial* configuration of actors is well formed, we define the well-formed relation, *wf*, inductively over elements of $\mathbb{F}$ as follows:

$$wf\,(F) = \begin{cases} wf\,(F_1) \wedge wf\,(F_2) & \text{if } F = F_1\,,\,F_2 \\ wf\,(F_1) & \text{if } F = [\,F_1\,]\lceil\{\,R\,\} \\ \textbf{false} & \text{if } F = p\,(\,a\,,\,b\,|\,\sigma : [s]\ \lambda : [l]\ \tau : [t]) \wedge t \neq \varsigma \\ \textbf{false} & \text{if } F = r \blacktriangleleft (s, e, v) \wedge r \neq \varsigma \wedge s \neq \varsigma \\ \textbf{true} & \text{otherwise} \end{cases}$$

We may then restate the configuration operation as follows:

$\langle\_\rangle : \mathbb{F} \overset{\circ}{\longrightarrow} \mathbb{C}$

- Defines a configuration where $\langle F \rangle \in \mathbb{C}$ if $wf\,(F)$.

4.2.3 Meta Architecture

We define a *meta architecture* for actors using the constraints given in Section 2.2. Specifically, a *meta-actor* is an actor which is capable of processing signals and generating notifications for other actors. Under this definition, a meta-actor may customize the behavior of a particular system service. For example, a meta-actor may intercept **transmit** events generated by another actor in order to customize message passing. Actors which have their events intercepted in this fashion are called *base actors.*

We formalize meta-level customization as follows. We add the **install** signal to the sort of events and define the *block* relation as shown in Figure 4.4. We also introduce two new sorts: the sort of signals, $\mathbb{S}ig \subset \mathbb{E}$, and the sort of notifications, $\mathbb{N}ot \subset \mathbb{E}$, with:

$$\mathbb{S}ig = \{\textbf{transmit}, \textbf{create}, \textbf{ready}, \textbf{install}\}$$

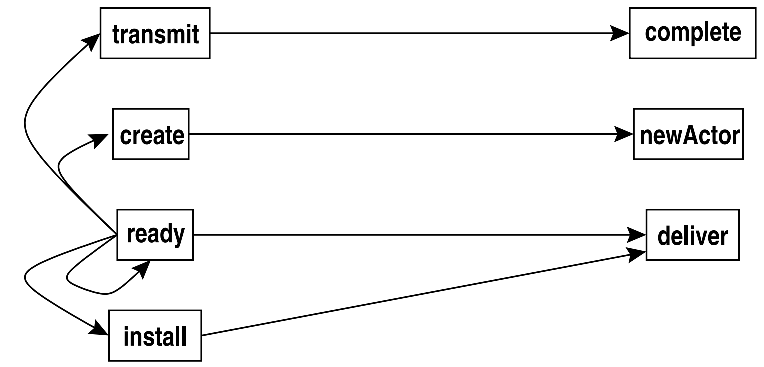


Figure 4.4: **Meta Actor Transitions:** The *block* relation for an actor semantics which support meta-level customization.

$$\mathbb{N}ot = \{\textbf{complete}, \textbf{newActor}, \textbf{deliver}\}$$

Actor computation is slightly modified to allow meta-level customization. Specifically, we must route the events generated by meta-actors so that signals and notifications are handled correctly. Moreover, in the case where a meta-actor is blocked on a **ready**, we still want to allow the meta-actor to process any signals generated by its base actor. We enable meta-actor computation by replacing the `request` rule with two sub-rules which handle signals and notifications separately:

`[request-meta]`

$$!\,(\,a\,,\,b\,|\,\sigma:[s]\;\lambda:[l]\;\tau:[t])\;\longrightarrow\;?\,(\,a\,,\,b\,|\,\sigma:[s]\;\lambda:[e]\;\tau:[t])\,,\,t\blacktriangleleft(a,e,v)$$

$$\text{if}\quad \begin{array}{ll} \mathit{nextEvent}(s,b) & =(e,v) \quad \text{and} \\ e & \in \mathbb{S}ig \end{array}$$

`[request-base]`

$$!\,(\,a'\,,\,b'\,|\,\sigma:[s']\;\lambda:[l']\;\tau:[t'])\,,\,p\,(\,a\,,\,b\,|\,\sigma:[s]\;\lambda:[l]\;\tau:[a'])\;\longrightarrow$$
$$?\,(\,a'\,,\,b'\,|\,\sigma:[s']\;\lambda:[l']\;\tau:[t'])\,,\,p\,(\,a\,,\,b\,|\,\sigma:[s]\;\lambda:[l]\;\tau:[a'])\,,\,a\blacktriangleleft(a',e,v)$$

$$\text{if}\quad \begin{array}{ll} \mathit{nextEvent}(s',b') & =(e,v) \quad \text{and} \\ e & \in \mathbb{N}ot \end{array}$$

The request rules correctly route events depending on whether they are signals or notifications. Note that, in the case of a meta-actor sending a notification, the “last event generated” attribute

is not set to the notification generated. Thus, after sending a notification, a meta-actor blocks on whatever signal it was previously blocked on before receiving the request from its base actor. By the construction of *block*, this signal will always be **ready**. This slightly unusual formulation of the rules has the pleasant side-effect of allowing a meta-actor to return to its previous blocked context without requiring extra bookkeeping in the rewrite terms.

We introduce the `install`, `redirect` and `propagate` rules to handle meta-actor installation and message redirection, respectively:

`[install]`

$$\begin{array}{l} A\,,\,A'\,,\,\varsigma \blacktriangleleft (a,\mathbf{install},\{a',b,v\})\,,\,\varsigma \blacktriangleleft (a',\mathbf{ready},\{\}) \quad\longrightarrow \\ \qquad\qquad [\,A\,,\,A'_*\,,\,A''\,,\,a \blacktriangleleft (\varsigma,\mathbf{newActor},\{a''\})\,,\,a'' \blacktriangleleft (a',\mathbf{ready},\{\})\,]\,\lceil\{\,a,a'\,\} \end{array}$$

$$\begin{array}{llll} \text{where} & A & = ?\,(\,a\,,\,b_a\,|\,\sigma : [s_a]\;\lambda : [\mathbf{install}]\;\tau : [t_a]) & \text{and} \\ & A' & = ?\,(\,a'\,,\,b_{a'}\,|\,\sigma : [s_{a'}]\;\lambda : [\mathbf{ready}]\;\tau : [\varsigma]) & \text{and} \\ & A'_* & = ?\,(\,a'\,,\,b_{a'}\,|\,\sigma : [s_{a'}]\;\lambda : [\mathbf{ready}]\;\tau : [a'']) & \text{and} \\ & A'' & = ?\,(\,a''\,,\,b\,|\,\sigma : [s]\;\lambda : [\mathbf{ready}]\;\tau : [\varsigma]) & \text{and} \\ & \mathit{newInstance}(b,v) & = s & \end{array}$$

`[redirect]`

$$p\,(\,a\,,\,b\,|\,\sigma : [s]\;\lambda : [l]\;\tau : [t])\,,\,a : a' \triangleleft v \quad\longrightarrow\quad p\,(\,a\,,\,b\,|\,\sigma : [s]\;\lambda : [l]\;\tau : [t])\,,\,a : a' \triangleleft (\mathbf{rcv},a,v)$$

$$\begin{array}{llll} \text{if} & t & \neq \varsigma & \text{and} \\ & v & \neq (\mathbf{rcv},x,y) & \end{array}$$

`[propagate]`

$$\begin{array}{l} p\,(\,a\,,\,b\,|\,\sigma : [s]\;\lambda : [l]\;\tau : [t])\,,\,a : a' \triangleleft (\mathbf{rcv},a^*,v) \quad\longrightarrow \\ \qquad p\,(\,a\,,\,b\,|\,\sigma : [s]\;\lambda : [l]\;\tau : [t])\,,\,t : a' \triangleleft (\mathbf{rcv},a^*,v) \end{array}$$

$$\text{if}\;\; t \;\neq \varsigma$$

The `install` rule creates a new meta-actor and installs it on a particular base actor[5]. The installation is only performed when the base actor is blocked on a **ready** signal. Moreover, the structure of the rule implies that an actor may never install a meta-actor on itself. During an `install`, the transition map is updated so that all events generated by the base actor are intercepted by the new meta-actor. The `redirect` rule constructs a `rcv` message out of a message targeted to a base actor. A `rcv` message is propagated to the appropriate meta-actor

[5] The restriction on the right side of the rule serves the same purpose as the restriction on the `create` rule. Namely, it ensures that fragment compositions are well-formed.

by the `propagate` rule. The second parameter in a `rcv` message gives the name of the original target of the message.

As noted in Section 2.2, our meta-level model is parameterized by the choice of events and the structure of the *block* relation. Alternative parameterizations, such as that described in Section 2.2.2, may be specified by redefining the block relation and the actor rules given above.

4.3 A Semantics for Software Architectures

In Section 2.3, we defined an *actor group* as an encapsulated collection of actors with special operations for allowing interactions between groups. In this section, we extend the rewriting semantics with support for group operations.

A rewriting semantics with group support adds three properties to an actor configuration:

- **Group Membership:** Each actor has a modifiable *group membership* which indicates the groups an actor belongs to. The group an actor is created in is called the actor's *initial* group. A new actor has the same initial group as its creator. Note that group membership is monotonic. That is, once admitted, an actor is never removed from a group.

- **Encapsulated Interactions:** Actors may interact only with other actors in a common group. Encapsulation properties are enforced at the time a message is sent. That is, a **transmit** will not be transformed into a message unless the sender and receiver are a member of a common group.

- **Managers:** Groups are instantiated with a single member actor called the *group manager*. Managers are exempt from the interaction restrictions applied to regular actors. Moreover, managers are the only actors that can alter group membership, create new external groups, or install customizations on internal actors.

While groups are normally isolated from one another, there are three mechanisms for allowing interactions between groups:

- **Manager Interactions:** A group manager may receive a message from any other actor in the system (including other group managers). Managers coordinate admission and meta-actor installation.

- **Admission:** A group manager can admit an actor to its group by changing the actor's group membership. Once admitted, the actor can communicate with any actor in the admitting group.

- **Meta Actor Installation:** A group manager can customize a group member by installing a new meta-actor. The new meta-actor is created as a member of an external group. Thereafter, the new meta-actor will intercept any signals generated by the base actor. A meta-actor inherits the group membership of the actor it customizes.

We support groups in our semantics by adding group membership attributes to actors. Specifically, we model actors as 9-tuples, $(p, a, b, s, l, t, i, g, r)$, with attributes:

$p \in \mathbb{P}$	A processing state.
$a \in \mathbb{A} \setminus \{\varsigma\}$	An address.
$b \in \mathbb{B}$	A behavior.
$s \in \mathbb{S}$	A state.
$l \in \mathbb{E}$	The last event generated by an actor.
$t \in \mathbb{A}$	The transition map for an actor.
$i \in \mathbb{A}$	The initial group of an actor.
$g \in \mathbf{P}_\omega[\mathbb{A}]$	The group membership of an actor.
$r \in \mathbb{A}$	The base map of an actor.

Actors are represented by terms of the form:

$$p\,(\,a\,,\,b\,,\,i\,,\,r\,|\,\sigma : [s]\ \lambda : [l]\ \tau : [t]\ \gamma : [g])$$

where the initial group and base map, i and r respectively, are fixed attributes of an actor, whereas group membership, g, is a run-time attribute. Each group is referred to by the name of its manager. The initial group, i, gives the group an actor was created in. In particular, an actor is a group manager if $a = i$. We use this information to determine group membership of actors created at run-time. Group membership, g, gives the groups to which an actor belongs.

The base map, r, indicates if this actor is a meta-actor in a meta-level stack. Specifically, an actor is a base actor if $a = r$. Otherwise, the actor is a meta-actor with r giving the name of the base actor in the stack. The base map is used to determine the group membership of a new actor created by a meta-actor.

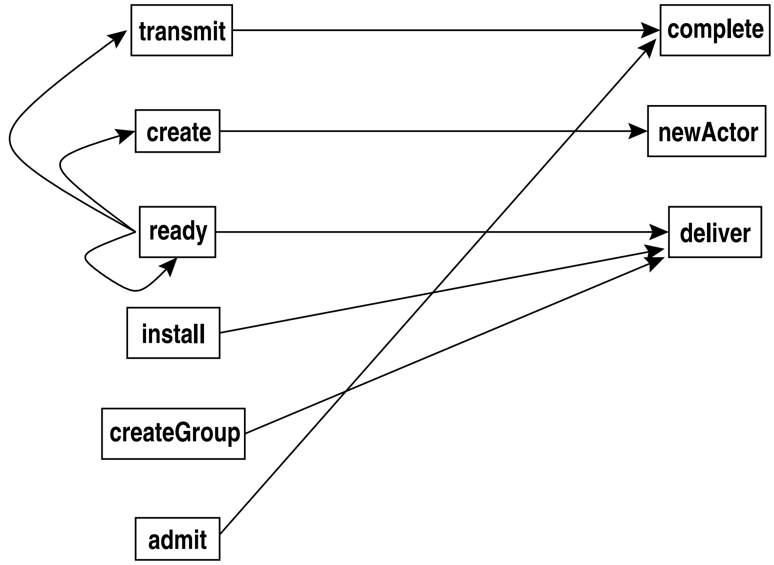


Figure 4.5: **Group Transitions:** The *block* relation for an actor semantics with group support.

We add two new signals to the semantics, **createGroup** and **admit**, and define the *block* relation as shown in Figure 4.5. Although signals can be generated by any actor, the **install**, **createGroup** and **admit** signals will be processed by the system only if they originate from managers. We also enforce the constraint that managers may never be created as meta-actors. Similarly, managers may not be customized by meta-actors. Thus, *block* need not include links from **ready** to **install**, **createGroup** and **admit**.

4.3.1 Actor Operations

The rules governing basic actor operations are slightly altered to support group operations. In particular, only the `send` and `create` rules must be changed:

`[send-in]`

$$A\,,\,A'\,,\,\varsigma \blacktriangleleft (a, \mathbf{transmit}, \{a', v\}) \quad\longrightarrow\quad A\,,\,A'\,,\,a \blacktriangleleft (\varsigma, \mathbf{complete}, \{\})\,,\,a' \lhd v$$

$$\begin{array}{rrll} \text{if} & A & = p_a\,(\,a\,,\,b_a\,,\,i_a\,,\,r_a \,|\, \sigma : [s_a]\ \lambda : [l_a]\ \tau : [t_a]\ \gamma : [g_a]) & \text{and} \\ & A' & = p_{a'}\,(\,a'\,,\,b_{a'}\,,\,i_{a'}\,,\,r_{a'} \,|\, \sigma : [s_{a'}]\ \lambda : [l_{a'}]\ \tau : [t_{a'}]\ \gamma : [g_{a'}]) & \text{and} \\ & \multicolumn{2}{l}{((g_a \cap g_{a'} \neq \emptyset) \vee (a' = i_{a'}))} & \end{array}$$

`[send-out]`

$$\langle F\,,\,A\,,\,\varsigma \blacktriangleleft (a, \mathbf{transmit}, \{a', v\})\rangle \quad\longrightarrow\quad \langle F\,,\,A\,,\,a \blacktriangleleft (\varsigma, \mathbf{complete}, \{\})\,,\,a' \lhd v\rangle$$

$$\begin{array}{rrll} \text{if} & A & = p_a\,(\,a\,,\,b_a\,,\,a\,,\,r_a \,|\, \sigma : [s_a]\ \lambda : [l_a]\ \tau : [t_a]\ \gamma : [g_a]) & \text{and} \\ & a' & \notin \mathit{recep}(F) & \end{array}$$

`[create-base]`

$$A\,,\,\varsigma \blacktriangleleft (a, \mathbf{create}, \{b', v\}) \quad\longrightarrow\quad [\,A\,,\,a \blacktriangleleft (\varsigma, \mathbf{newActor}, \{a'\})\,,\,A'\,,\,\varsigma \blacktriangleleft (a', \mathbf{ready}, \{\})\,]\lceil\{\,a\,\}$$

$$\begin{array}{lrll} \text{if} & A & = ?\,(\,a\,,\,b\,,\,i\,,\,a \,|\, \sigma : [s]\ \lambda : [\mathbf{create}]\ \tau : [\varsigma]\ \gamma : [g]) & \text{and} \\ & A' & = ?\,(\,a'\,,\,b'\,,\,i\,,\,a' \,|\, \sigma : [s']\ \lambda : [\mathbf{ready}]\ \tau : [\varsigma]\ \gamma : [\{i\}]) & \text{and} \\ & \mathit{newInstance}(b', v) & = s' & \text{and} \\ & a' & \notin \mathit{acq}(s) \cup \{a\} \cup \mathit{acq}(v) & \end{array}$$

`[create-meta]`

$$\begin{array}{l} A_b\,,\,A\,,\,\varsigma \blacktriangleleft (a, \mathbf{create}, \{b', v\}) \quad\longrightarrow \\ \qquad\qquad [\,A_b\,,\,A\,,\,a \blacktriangleleft (\varsigma, \mathbf{newActor}, \{a'\})\,,\,A'\,,\,\varsigma \blacktriangleleft (a', \mathbf{ready}, \{\})\,]\lceil\{\,a, a_b\,\} \end{array}$$

$$\begin{array}{lrll} \text{if} & A_b & = p_b\,(\,a_b\,,\,b_b\,,\,i_b\,,\,a_b \,|\, \sigma : [s_b]\ \lambda : [l_b]\ \tau : [t_b]\ \gamma : [g_b]) & \text{and} \\ & A & = ?\,(\,a\,,\,b\,,\,i\,,\,a_b \,|\, \sigma : [s]\ \lambda : [\mathbf{create}]\ \tau : [\varsigma]\ \gamma : [g]) & \text{and} \\ & A' & = ?\,(\,a'\,,\,b'\,,\,i_b\,,\,a' \,|\, \sigma : [s']\ \lambda : [\mathbf{ready}]\ \tau : [\varsigma]\ \gamma : [\{i_b\}]) & \text{and} \\ & \mathit{newInstance}(b', v) & = s' & \text{and} \\ & a' & \notin \mathit{acq}(s) \cup \{a\} \cup \mathit{acq}(v) & \end{array}$$

The `send` rule is divided into two cases depending on whether or not the target actor is a member of the configuration. The `send-in` rule enforces the constraint that actors within a configuration may exchange messages only if they are a member of a common group. Note that group membership is an additive property (*i.e.* actors may never be removed from their groups), thus encapsulation need be enforced only within the `send` rule. The condition $g_a \cap g_{a'} \neq \emptyset$ holds if the sender and receiver are a member of a common group. The condition $a' = i_{a'}$ holds if the receiver is a group manager. Thus, the send is allowed if either both actors are a member of a common group, or if the receiver is a group manager. The `send-out` rule allows a manager to send a message to a target outside the configuration (see the `in` rule below).

The `create` rule is subdivided into two cases depending on whether or not the creator is a meta-actor. We make this division to preserve the constraint that the new actor has the same initial group as the base actor in the meta-level stack of the creator. We enforce this constraint to avoid ambiguity in the case of an actor which is customized by a meta-actor. In particular, when a meta-actor generates a `create` request, it may be doing so on its own behalf or on behalf of its base actor. Thus there are two valid choices for the initial group of the new actor: the initial group of the meta-actor, or the initial group of the base actor.

The `create-base` rule handles the case where an uncustomized actor performs the create. In this case, the new actor is added to the same initial group as the creator. The `create-meta` rule handles the case where a meta-actor performs the create. In this case, the new actor is added to the initial group of the base actor, a_b. As with any create, the address of the new actor is returned to the creator, and the resulting term is restricted to preserve subsequent fragment compositions.

Only the `in` rule needs to be changed for handling external interactions:

`[in]`

$$\langle F \rangle \quad \longrightarrow \quad \langle F \, , \, a \lhd v \rangle$$

$$\begin{array}{rrll} \text{if} & F & = [\, F' \, , \, M \,] \lceil \{ \, R \, \} & \text{and} \\ & M & = p \, (\, a \, , \, b \, , \, a \, , \, a \, | \, \sigma : [s] \; \lambda : [l] \; \tau : [t] \; \gamma : [g]) & \text{and} \\ & a & \in \mathit{recep}(F) & \end{array}$$

The `in` rule is altered so that external messages are admitted only if they are targeted to a group manager which has been exported as a receptionist (*i.e.* group managers are the only valid receptionists for a configuration). Note that it is necessary to alter the `in` rule because the more general rule normally associated with actor semantics would allow group encapsulation to be violated[6].

[6] Specifically, an external message might be targeted to a non-manager actor.

4.3.2 Meta Actor Operations

Meta-actor installation is slightly modified under group semantics while the `redirect` rule is unchanged:

`[install]`

$$
\begin{array}{l}
A\,,\,A'\,,\,\varsigma \blacktriangleleft (a, \mathbf{install}, \{a', b'', v, g''\})\,,\,\varsigma \blacktriangleleft (a', \mathbf{ready}, \{\}) \;\longrightarrow \\
\qquad [\,A\,,\,A'_{*}\,,\,A''\,,\,a \blacktriangleleft (\varsigma, \mathbf{newActor}, \{a''\})\,,\,a'' \blacktriangleleft (a', \mathbf{ready}, \{\})\,]\,\lceil\{\,a, a'\,\}
\end{array}
$$

$$
\begin{array}{rrll}
\text{where} & A & = ?\,(\,a\,,\,b\,,\,a\,,\,a\,|\,\sigma : [s]\;\lambda : [\mathbf{install}]\;\tau : [\varsigma]\;\gamma : [g]) & \text{and} \\
 & A' & = ?\,(\,a'\,,\,b'\,,\,a\,,\,a_b\,|\,\sigma : [s']\;\lambda : [\mathbf{ready}]\;\tau : [\varsigma]\;\gamma : [g']) & \text{and} \\
 & A'_{*} & = ?\,(\,a'\,,\,b'\,,\,a\,,\,a_b\,|\,\sigma : [s']\;\lambda : [\mathbf{ready}]\;\tau : [a'']\;\gamma : [g']) & \text{and} \\
 & A'' & = ?\,(\,a''\,,\,b''\,,\,g''\,,\,a_b\,|\,\sigma : [s'']\;\lambda : [\mathbf{ready}]\;\tau : [\varsigma]\;\gamma : [g' \cup \{g''\}]) & \text{and} \\
 & a & \neq g'' &
\end{array}
$$

The `install` rule enforces the following constraints: 1) only group managers may request installation; 2) installation may be performed only on an actor in the requester's group; and 3) group managers may not be customized. The parameter g'' indicates the group where the new actor should be added. Note that the new actor is created in a group different from the requesting manager. This is the only case in which a group manager may affect the membership of an external group[7]. Note also that the new meta-actor inherits the group membership of its base actor. This is necessary so that any messages sent by the meta-actor on behalf of its base actor may be properly delivered.

4.3.3 Group Operations

Finally, we require three new rules which allow group managers to admit actors and create new external groups:

[7] We have structured the rule this way because we view installation as a more sensitive encapsulation property than membership. That is, we sacrifice membership encapsulation in favor of allowing managers ultimate control of customization of internal actors.

[admit-top]

$$A\,,\,A'\,,\,\varsigma \blacktriangleleft (a,\mathbf{admit},\{a'\}) \quad\longrightarrow\quad A\,,\,A'_*\,,\,a \blacktriangleleft (\varsigma,\mathbf{complete},\{\})$$

$$\begin{array}{rrll} \text{where} & A & = ?\,(\,a\,,\,b\,,\,a\,,\,a\,|\,\sigma:[s]\ \lambda:[\mathbf{admit}]\ \tau:[\varsigma]\ \gamma:[g]) & \text{and} \\ & A' & = p'\,(\,a'\,,\,b'\,,\,i'\,,\,r'\,|\,\sigma:[s']\ \lambda:[l']\ \tau:[\varsigma]\ \gamma:[g']) & \text{and} \\ & A'_* & = p'\,(\,a'\,,\,b'\,,\,i'\,,\,r'\,|\,\sigma:[s']\ \lambda:[l']\ \tau:[\varsigma]\ \gamma:[g'\cup\{a\}]) & \end{array}$$

[admit-base]

$$A\,,\,A'\,,\,\varsigma \blacktriangleleft (a,\mathbf{admit},\{a'\}) \quad\longrightarrow\quad A\,,\,A'_*\,,\,\varsigma \blacktriangleleft (a,\mathbf{admit},\{t'\})$$

$$\begin{array}{rrll} \text{where} & A & = ?\,(\,a\,,\,b\,,\,a\,,\,a\,|\,\sigma:[s]\ \lambda:[\mathbf{admit}]\ \tau:[\varsigma]\ \gamma:[g]) & \text{and} \\ & A' & = p'\,(\,a'\,,\,b'\,,\,i'\,,\,r'\,|\,\sigma:[s']\ \lambda:[l']\ \tau:[t']\ \gamma:[g']) & \text{and} \\ & A'_* & = p'\,(\,a'\,,\,b'\,,\,i'\,,\,r'\,|\,\sigma:[s']\ \lambda:[l']\ \tau:[t']\ \gamma:[g'\cup\{a\}]) & \text{and} \\ & t' & \neq \varsigma & \end{array}$$

[create-group]

$$A\,,\,\varsigma \blacktriangleleft (a,\mathbf{createGroup},\{b',v\}) \quad\longrightarrow\quad [\,A\,,\,A'\,,\,a \blacktriangleleft (\varsigma,\mathbf{newActor},\{a'\})\,,\,\varsigma \blacktriangleleft (a',\mathbf{ready},\{\})\,]\,\lceil\{\,a\,\}$$

$$\begin{array}{rrll} \text{where} & A & = ?\,(\,a\,,\,b\,,\,a\,,\,a\,|\,\sigma:[s]\ \lambda:[\mathbf{createGroup}]\ \tau:[\varsigma]\ \gamma:[g]) & \text{and} \\ & A' & = ?\,(\,a'\,,\,b'\,,\,a'\,,\,a'\,|\,\sigma:[s']\ \lambda:[\mathbf{ready}]\ \tau:[\varsigma]\ \gamma:[\{a'\}]) & \text{and} \\ & \mathit{newInstance}(b',v) & = s' & \text{and} \\ & a' & \notin \mathit{acq}(s)\cup\{a\}\cup\mathit{acq}(v) & \end{array}$$

The "admit" rules (admit-top and admit-base) allow group managers to change group membership by admitting other actors in the configuration. Note that actors external to the configuration can never be admitted to a group. We require two admit rules because, by definition, meta-actors inherit the group membership of their base actors. Thus, if a base actor is admitted, then the membership of its meta-actor must also be changed. The admit-top rule handles the default case where a top-level actor is admitted to a group. The admit-base rule handles the case where a customized actor is admitted to a group. In this case, the actor is admitted and the admit request is rewritten so that the meta-actor is also admitted. The create-group rule allows the creation of new groups. Groups are created by specifying a behavior for the manager of the group. Group creation is the only mechanism for creating group managers. Thus, all groups have only a single manager[8].

[8] Because managers are intended for coordinating connections between groups (*e.g.* by admitting external actors), the need for multiple managers in a single group is an implementation rather than semantic concern. That is, we may require multiple managers to allow a more efficient implementation (*e.g.* multi-threaded), although the behavior of these managers is semantically equivalent to a single manager.

4.4 Interaction Semantics

The semantics given in the previous section describes the underlying behavior of actor configurations, but provides little insight into more abstract group relationships. In particular, it is not apparent if we can determine the "compatibility" of the groups selected for a particular architecture. In saying that a collection of groups is "compatible", we imply that the following questions can be answered:

- Given actor groups g_1 and g_2:
  - Do the messages *sent* by actors in g_1 to actors in g_2 correspond to the messages *expected* by the actors in g_2 (and vice versa)?
  - Do the actors in g_1 which *customize* (*i.e.* are meta-actors for) the actors in g_2 preserve the semantics of the actors they customize? That is, are signals handled correctly and are appropriate notifications generated?

These conditions provide a natural analog to compatibility requirements we might expect from the structures introduced in Chapter 3. In particular, because modules, protocols and policies are mapped directly onto actor groups, we may use compatibility conditions to infer, for example, whether or not a particular protocol is an appropriate connector for a pair of modules. Similarly, we can infer whether or not a policy correctly customizes a protocol role or module.

To provide these compatibility conditions, we derive an interaction semantics based on the group semantics of the previous section. This semantics is based on the notion of *computation paths* entailed by a particular configuration. We summarize the definitions of paths here, a more complete definition may be found in [55].

4.4.1 Paths and Interaction Steps

The initial model construction (see [41]) of a rewrite system, (Σ, E, L, R) yields a set of finite computations, P. Each such computation may be viewed as a sequence of transitions where each element of the sequence is a term, and each transition represents the application of a rule.

The initial model construction also provides an equivalence relation, $\sim$, on computations in P. By construction, $\sim$ satisfies (Σ, E), as well as certain categorical and functoriality constraints. More importantly, $\sim$ also satisfies "exchange" laws of the form:

$$\frac{\alpha_1 : [w_1] \rightarrow [w_1'] \;\; \ldots \;\; \alpha_n : [w_n] \rightarrow [w_n']}{r(\bar{\alpha}) \;\; = \;\; r([\bar{w}]); t'(\bar{\alpha}) \;\; = \;\; t(\bar{\alpha}); r([\bar{w'}])}$$

for each rule $r : [t(\bar{x})] \rightarrow [t'(\bar{x})]$. This rule allows the "serialization" of concurrent computations and may be used to flatten a concurrent computation into a sequence of atomic rewrite steps.

Following [55], we define a *computation path*, $p : \aleph \rightarrow P$, as an infinite sequence derived from P such that adjacent computations are sequentially composable[9]. We use $Path[P]$ to denote the set of all such paths over P. The expression $p\lceil i$ denotes the sequential composition of the first i segments of the path p (*i.e.* $p(0); p(1); \ldots; p(i)$). Finally, we define path equivalence by lifting the equivalence relation, $\sim$, to paths as follows:

Definition 1 (Path Equivalence) *For $p, p' \in Path[P]$, $p \sim p'$ if for all i, there exist k, c such that:*

$$p\lceil i; c \quad \sim \quad p'\lceil(i + k)$$

and for all i', there exist k', c' such that:

$$p'\lceil i'; c' \quad \sim \quad p\lceil(i' + k')$$

Our goal in defining an interaction semantics is to isolate the external, group-to-group interactions entailed by a particular actor configuration. To this end, we introduce the *partial configuration* operator:

$$\langle \_ \rangle \, [\_] \, \{\_\} : \mathbb{F} \times (\mathbf{P}_\omega[\mathbb{A}] \times \mathbf{P}_\omega[\mathbb{A}]) \times \mathbf{P}_\omega[\mathbb{A}] \longrightarrow \mathbb{C}$$

where $\langle A \rangle \, [b] \, \{m\} \in \mathbb{C}$ if:

- $\mathit{recep}(A) \cap (b^* \cup m) = \emptyset$ where $b^* = \{b_i | (b_i, a_i) \in b\}$;

- if $(b, a_1), (b, a_2) \in b$, then $a_1 = a_2$; and

[9] As in the previous section, we restrict ourselves to the fair paths derived from a particular rewrite theory.

- if $(b_1, a), (b_2, a) \in b$, then $b_1 = b_2$.

The term A represents an actor fragment which may or may not satisfy *wf* (*i.e.* may not be well formed). The relation b is referred to as the *base relation* for the configuration, while the set m is referred to as the *member set*.

The configuration $M = \langle A \rangle \, [b] \, \{m\}$ is partial in the sense that it may not include all the actors referred to by elements of the fragment A. The member set, m, is interpreted as the set of external members of the configuration. That is, an address a is included in m if the actor it represents is a member of a group in A, but is not contained in A itself. Similarly, the base relation, b, is interpreted as the set of external base actors of the configuration. An element of the base set, $(b_i, a_i) \in b$, indicates a base actor with address b_i with meta-actor a_i. The constraints defined above restrict the base relation so that base actors have at most one meta-actor, and vice versa.

We use the members relation, $\mathit{members} : \mathbb{F} \longrightarrow \mathbf{P}_\omega[\mathbb{A}]$, and the meta-set relation, $\mathit{metaSet} : \mathbb{F} \longrightarrow \mathbf{P}_\omega[\mathbb{A}]$, to refer to particular addresses referenced in a fragment. In particular, the members relation gives the set of actors contained within a fragment. The meta-set relation is defined as:

$$\mathit{metaSet}(A) = \{t \mid p\,(\,a\,,\,b\,,\,i\,,\,a_b \,|\, \sigma : [s] \;\lambda : [l] \;\tau : [t] \;\gamma : [g]) \in A\}$$

where $A \in \mathbb{F}$. That is, the meta-set relation gives the address of every actor which customizes an actor in A.

Finally, we define the set of interaction steps as a collection of concurrent rewrites over partial configurations. In the following definitions, we use g to denote the set of concurrent rewrites defined in the previous sections, excluding `install`, `admit`, `in` and `out`. The set of interaction steps is defined as follows:

`[silent()]`

$$\langle A \rangle \, [b] \, \{m\} \implies \langle A' \rangle \, [b] \, \{m\}$$

$$\text{if } A \xrightarrow{g} A'$$

`[emit-1(a,a’,e,v)]`

$$\big\langle A\,,\,a' \blacktriangleleft (a,e,v)\big\rangle\,[b]\,\{m\} \implies \langle A\rangle\,[b]\,\{m\}$$

$$\text{if}\quad a' \quad \notin \mathit{members}(A)$$

`[emit-2(a,a’,v)]`

$$\big\langle A\,,\,a : a' \lhd v\big\rangle\,[b]\,\{m\} \implies \langle A\rangle\,[b]\,\{m\}$$

$$\text{if}\quad a \quad \notin \mathit{members}(A)$$

`[consume-1(a,a’,e,v)]`

$$\langle A\rangle\,[b]\,\{m\} \implies \big\langle A\,,\,a \blacktriangleleft (a',e,v)\big\rangle\,[b]\,\{m\}$$

$$\begin{array}{rrll} \text{if} & a & \in \mathit{members}(A) & \text{and} \\ & (a',a) \in b & \text{or } a' \in \mathit{metaSet}(A) \setminus \{\varsigma\} & \end{array}$$

`[consume-2(a,a’,v)]`

$$\langle A\rangle\,[b]\,\{m\} \implies \big\langle A\,,\,a : a' \lhd v\big\rangle\,[b]\,\{m\}$$

$$\begin{array}{rrll} \text{if} & a & \in \mathit{members}(A) & \text{and} \\ & a' & \in m & \end{array}$$

`[admit(a’)]`

$$\big\langle F\,,\,A\,,\,\varsigma \blacktriangleleft (a,\mathbf{admit},\{a'\})\big\rangle\,[b]\,\{m\} \implies \langle F\,,\,A\,,\,a \blacktriangleleft (\varsigma,\mathbf{complete},)\rangle\,[b]\,\big\{m \cup \{a'\}\big\}$$

$$\begin{array}{rrll} \text{if} & A & = ?\,(\,a\,,\,b\,,\,a\,,\,a\,|\,\sigma : [s]\ \lambda : [\mathbf{admit}]\ \tau : [\varsigma]\ \gamma : [g]) & \text{and} \\ & a' & \notin \mathit{members}(F\,,\,A) & \end{array}$$

`[install-1(a’’)]`

$$\begin{array}{l} \langle F\,,\,A\,,\,A'\,,\,\varsigma \blacktriangleleft (a,\mathbf{install},\{a',b'',v,g\})\rangle\,[b]\,\{m\} \implies \\ \qquad \langle F\,,\,A\,,\,A'_*\,,\,a \blacktriangleleft (\varsigma,\mathbf{newActor},\{a''\})\rangle\,[b]\,\{m \cup \{a''\}\} \end{array}$$

$$\begin{array}{rrll} \text{where} & A & = ?\,(\,a\,,\,b\,,\,a\,,\,a\,|\,\sigma : [s]\ \lambda : [\mathbf{install}]\ \tau : [\varsigma]\ \gamma : [g_a]) & \text{and} \\ & A' & = ?\,(\,a'\,,\,b'\,,\,a\,,\,a'_b\,|\,\sigma : [s']\ \lambda : [\mathbf{ready}]\ \tau : [\varsigma]\ \gamma : [g']) & \text{and} \\ & A'_* & = ?\,(\,a'\,,\,b'\,,\,a\,,\,a'_b\,|\,\sigma : [s']\ \lambda : [\mathbf{ready}]\ \tau : [a'']\ \gamma : [g']) & \text{and} \\ & g & \notin \mathit{members}(F\,,\,A) & \text{and} \\ & a'' & \notin \mathit{members}(F\,,\,A\,,\,A') & \end{array}$$

$\texttt{[install-2(a,a}_{\texttt{b}}\texttt{,g}_{\texttt{b}}\texttt{)]}$

$$\langle F\,,\,A_m\rangle\,[b]\,\{m\} \implies \langle F\,,\,A_m\,,\,?\,(\,a\,,\,b\,,\,a_m\,,\,g_b\,|\,\sigma : [s]\ \lambda : [\mathbf{ready}]\ \tau : [\varsigma]\ \gamma : [g])\rangle\,[b \cup \{a_b\}]\,\{m \cup \{a_b\}\}$$

$$\begin{array}{rrll} \text{if} & g_b, a & \notin \mathit{members}(A) & \text{and} \\ & A_m & = p_m\,(\,a_m\,,\,b_m\,,\,a_m\,,\,a_m\,|\,\sigma : [s_m]\ \lambda : [l_m]\ \tau : [\varsigma]\ \gamma : [g_m]) & \text{and} \\ & a_m & \in g & \end{array}$$

We use the symbol $\mathcal{I}_g$ to denote the set of interaction steps: `silent`, `emit-1`, `emit-2`, `consume-1`, `consume-2`, `admit`, `install-1`, and `install-2`. Let γ range over $\mathcal{I}_g$.

Definition 2 (Computation Paths) *A computation path, π, is an infinite sequence of interaction steps with $\pi(i) = \gamma_i : \langle A_i \rangle \, [b_i] \, \{m_i\} \Rightarrow \langle A'_i \rangle \, [b'_i] \, \{m'_i\}$ such that $A_{i+1} \sim A'_i$ for $i \in \aleph$. Note that for each computation path, π, there exists a path $p \in Path[P]$ such that if $\pi(i) = \gamma_i : \langle A_i \rangle \, [b_i] \, \{m_i\} \Rightarrow \langle A'_i \rangle \, [b'_i] \, \{m'_i\}$, then $p \lceil i \sim A_i$. As a slight abuse of notation, we use the expression $Path(\langle A \rangle \, [b] \, \{m\})$ to denote the set of computation paths with source $\langle A \rangle \, [b] \, \{m\}$. Given a computation path, π, we define the function $members_i$ such that $members_i(\pi) = members(A_i)$ if $\pi(i) = \gamma_i : \langle A_i \rangle \, [b_i] \, \{m_i\} \Rightarrow \langle A'_i \rangle \, [b'_i] \, \{m'_i\}$.*

The computation paths entailed by a partial configuration may be viewed as a collection of "simulations" which describe the behavior of the configuration in response to various stimuli (*e.g.* interactions with external actors). Moreover, these paths define the expected behavior of an external stimulus for each particular scenario. In effect, we have derived a composition predicate for a partial configuration. We formalize this notion in the next section.

4.4.2 Composability

We derive composability criteria by giving an interaction semantics to partial configurations. This semantics is given by the following definitions:

Definition 3 (Interaction Sequences and Paths) *Let $\mathcal{I}_g^\infty = \{f \mid f : \aleph \rightarrow \mathcal{I}_g\}$ be the set of all interaction sequences, and let ζ range over $\mathcal{I}_g^\infty$. An interaction path is a 4-tuple, (A, b, m, ζ) where $\langle A \rangle \, [b] \, \{m\} \in \mathbb{C}$, and $\zeta \in \mathcal{I}_g^\infty$.*

Definition 4 (Sequence Dual) *Given an interaction sequence ζ, the dual of ζ, $\mathcal{D}(\zeta)$, is the interaction sequence ζ' defined as follows:*

- *If $\zeta(i) =$ `emit-1(a,a',e,v)`, then $\zeta'(i) =$ `consume-1(a',a,e,v)`.*
- *If $\zeta(i) =$ `emit-2(a,a',v)`, then $\zeta'(i) =$ `consume-2(a',a,v)`.*
- *If $\zeta(i) =$ `consume-1(a,a',e,v)`, then $\zeta'(i) =$ `emit-1(a',a,e,v)`.*
- *If $\zeta(i) =$ `consume-2(a,a',v)`, then $\zeta'(i) =$ `emit-2(a',a,v)`.*
- *Otherwise $\zeta'(i) =$ `silent()`.*

Definition 5 (Interaction Semantics) *The interaction semantics of a partial configuration, $\mathbb{I}(\langle A \rangle \, [b] \, \{m\})$, is the set of interaction paths abstracted from its fair computation paths:*

$$\mathbb{I}(\langle A \rangle \, [b] \, \{m\}) = \{tc(\pi) \mid \pi \in Path(\langle A \rangle \, [b] \, \{m\})\}$$

where $tc(\pi) = (A, b, m, \zeta)$ such that if $\pi(i) = \gamma_i : \langle A_i \rangle \, [b_i] \, \{m_i\} \Rightarrow \langle A'_i \rangle \, [b'_i] \, \{m'_i\}$, then $\zeta(i) = \gamma_i$.

Definition 6 (Interaction Equivalence) *Two partial configurations are interaction equivalent,* $\overset{\iota}{\sim}$*, just if they have the same interaction semantics:*

$$\langle A_1\rangle\,[b_1]\,\{m_1\} \;\overset{\iota}{\sim}\; \langle A_2\rangle\,[b_2]\,\{m_2\} \;\Leftrightarrow\; \mathbb{I}(\langle A_1\rangle\,[b_1]\,\{m_1\}) = \mathbb{I}(\langle A_2\rangle\,[b_2]\,\{m_2\})$$

In considering whether two partial configurations are composable, we are only concerned with those interactions which involve participants in both configurations. We define the *configuration image* of one configuration relative to another as follows:

Definition 7 (Configuration Image) *Given two partial configurations,* $\langle A_1\rangle\,[b_1]\,\{m_1\}$ *and* $\langle A_2\rangle\,[b_2]\,\{m_2\}$*, the configuration image,* $cimage_{\langle A_1\rangle[b_1]\{m_1\}}(\langle A_2\rangle\,[b_2]\,\{m_2\})$*, of* $\langle A_1\rangle\,[b_1]\,\{m_1\}$ *on* $\langle A_2\rangle\,[b_2]\,\{m_2\}$ *is defined as:*

$$cimage_{\langle A_1\rangle[b_1]\{m_1\}}(\langle A_2\rangle\,[b_2]\,\{m_2\}) = \{im_{\langle A_2\rangle[b_2]\{m_2\}}(\pi) \mid \pi \in \mathbb{I}(\langle A_1\rangle\,[b_1]\,\{m_1\})\}$$

where $im_{\langle A_2\rangle[b_2]\{m_2\}}(\pi) = \pi'$ *such that* π' *is the subset of interaction steps in* π *which refer to an actor contained in one of the states entailed by* $\mathbb{I}(\langle A_2\rangle\,[b_2]\,\{m_2\})$*. In particular, if* $\pi = (A_1, b_1, m_1, \zeta_1)$*, then* $\pi' = (A_1, b_1, m_1, \zeta_1')$ *where:*

- *If* $\zeta_1(i) =$ `emit-1(a,a',e,v)` *and* $a' \in members_j(\pi^*)$ *for some* $\pi^* \in \mathbb{I}(\langle A_2\rangle\,[b_2]\,\{m_2\})$*, then* $\zeta_1'(i) = \zeta(i)$*.*
- *If* $\zeta_1(i) =$ `emit-2(a,a',v)` *and* $a \in members_j(\pi^*)$ *for some* $\pi^* \in \mathbb{I}(\langle A_2\rangle\,[b_2]\,\{m_2\})$*, then* $\zeta_1'(i) = \zeta(i)$*.*
- *If* $\zeta_1(i) =$ `consume-1(a,a',e,v)` *and* $a' \in members_j(\pi^*)$ *for some* $\pi^* \in \mathbb{I}(\langle A_2\rangle\,[b_2]\,\{m_2\})$*, then* $\zeta_1'(i) = \zeta(i)$*.*
- *If* $\zeta_1(i) =$ `consume-2(a,a',v)` *and* $a' \in members_j(\pi^*)$ *for some* $\pi^* \in \mathbb{I}(\langle A_2\rangle\,[b_2]\,\{m_2\})$*, then* $\zeta_1'(i) = \zeta(i)$*.*
- *Otherwise,* $\zeta_1'(i) =$ `silent()`*.*

We define *configuration compatibility*, by considering the image of one configuration on another:

Definition 8 (Configuration Compatibility) *The partial configuration* $\langle A_1\rangle\,[b_1]\,\{m_1\}$ *is compatible with the partial configuration* $\langle A_2\rangle\,[b_2]\,\{m_2\}$*, denoted as:*

$$\langle A_1\rangle\,[b_1]\,\{m_1\} \overset{\iota}{\Rightarrow} \langle A_2\rangle\,[b_2]\,\{m_2\}$$

if for all $\pi \in cimage_{\langle A_1\rangle[b_1]\{m_1\}}(\langle A_2\rangle\,[b_2]\,\{m_2\})$ *with* $\pi = (A_1, b_1, m_1, \zeta_1)$*, there exists* $\pi' \in \mathbb{I}(\langle A_2\rangle\,[b_2]\,\{m_2\})$ *with* $\pi' = (A_2, b_2, m_2, \zeta_2)$ *such that* $\zeta_1 \sim \mathcal{D}(\zeta_2)$*.*

Finally, we define configuration composability as two partial configurations which are mutually compatible:

Definition 9 (Configuration Composability) *Two partial configurations,* $\langle A_1 \rangle\,[b_1]\,\{m_1\}$ *and* $\langle A_2 \rangle\,[b_2]\,\{m_2\}$*, are composable, with new partial configuration:*

$$\langle A_1\,,\,A_2 \rangle\,[(b_1 \cup b_2) \setminus L]\,\{(m_1 \cup m_2) \setminus L\}$$

where $L = members(A_1) \cup members(A_2)$*, just if* $members(A_1) \cap members(A_2) = \emptyset$ *and:*

$$\langle A_1 \rangle\,[b_1]\,\{m_1\} \overset{\iota}{\Rightarrow} \langle A_2 \rangle\,[b_2]\,\{m_2\} \quad \wedge \quad \langle A_2 \rangle\,[b_2]\,\{m_2\} \overset{\iota}{\Rightarrow} \langle A_1 \rangle\,[b_1]\,\{m_1\}$$

4.5 Summary

In this chapter, we have provided a formal semantics for the model described in Chapter 2. We first described a general actor semantics based on concurrent rewriting. This semantics was refined and specialized first to actors which may be customized by meta-actors, and then to a meta-architecture with actor group constraints. Finally, we provided an *interaction semantics* which considers only the interactions between partial configurations of actors.

We used our interaction semantics as basis for addressing composability issues in software architectures. Specifically, we provided a definition of composability based on whether or not two configurations are "compatible". Compatibility, in this case, refers to a correspondence between external interactions in each configuration.

Although our definition of composability is stronger than the usual definition associated with actor configurations (*cf.* [5]), relative to software architecture our definition is weak in the sense that it only ensures a correspondence between interactions in two partial configurations. Specifically, while our definition of composability guarantees that every "desirable" interaction pattern has a correspondence, we also guarantee that every "undesirable" pattern also has a correspondence. Thus, even if two partial configurations do not interact correctly (*i.e.* one or the other enters an undesirable state), they may still be composable.

In many ways, this definition is very similar to bisimilarity as defined in CCS [42] and π-calculus [43, 44]. In particular, it may be feasible to derive similar algebraic structures using partial configurations and composability. Moreover, it may be possible within this algebra to define stronger composability conditions. Such a definition would ensure that only the "desirable" interactions have a correspondence. We leave these issues as topics for future work.

Chapter 5

Conclusion

In this thesis, we have presented a new model for specifying software architectures. While traditional architectural styles may be specified within our model, we place particular emphasis on styles associated with distributed applications. Typically, distributed applications require architectural policies which enforce availability, dependability and other high-level constraints. Such policies are often called "cross-cutting" because their implementations may require access to internal component resources. Traditionally, these policies were implemented by hard-coding solutions within architectural elements, significantly reducing modularity.

A naive solution to this problem would be to arbitrarily expose component resources. However, this approach damages modularity in the same fashion as hard-coded implementations. In contrast, we augment our model with a *meta-architecture* which exposes resource utilization patterns. This meta-architecture preserves modularity by way of two key properties: transparency, and composability. The meta-architecture is transparent in the sense that base-level objects (*i.e.* architectural components) need not be aware that they are customized. Similarly, meta-level objects have limited access to the objects they customize: they may only respond to requests for resources. The meta-architecture is composable in the sense that a meta-object itself may be customized by another meta-object. As a result, multiple policies may be separately developed (as separate objects) and later composed on architectural elements.

Although we have based our model on a simplistic view of actor resources (*i.e.* those associated with the `send`, `create` and `ready` operations), we have illustrated that the model (and semantics) may be parameterized in order to address more specific issues. For example, an appropriate choice of events and rules may be used to address placement and load balancing issues, failure and recover, program visualization [11], or even real-time constraints [49].

To demonstrate the utility of the model, we have defined the *Distributed Connection Language* (DCL): an Actor-based architecture description language. DCL specifications may be used to define the initial configuration and dynamic restructuring of a distributed software architecture. In particular, DCL specifications may be used to define connections between architectural elements, as well as enforce high-level policies over collections of elements. The computational behavior of the architecture is defined by individual actors.

To demonstrate that the model may be implemented efficiently, we have described the mapping of DCL to the Actor Foundry, a Java-based actor programming environment. A key concern is the overhead entailed by the modularity of the system. Our performance tests show that DCL may be implemented with minimal overhead (less than 10%). Nonetheless, certain applications may require tighter performance bounds. To this end, we have provided alternative implementations which sacrifice certain assumptions associated with actor semantics for the sake of efficiency. Subsequent performance measurements indicate that overhead is reduced by a factor of two (*i.e.* less than 5%). Still other performance improvements are possible.

While specification and implementation are important issues, it is perhaps more important to derive tools which verify properties over specifications. Several properties are of interest. Does the interface of a component match the interface of the connector to which it is attached? Does the connector which attaches a pair of components implement the correct policy? Can one component be viewed as a refinement of another component? As a basis for answering such questions, we have developed a rewriting semantics for our model. Moreover, we have extended the semantics to the notion of a *partial configuration*, and have derived composability conditions which are based on the matching of interactions between partial configurations. A substantial challenge for future work is to translate this "weak" condition for composability into a stronger predicate which captures the properties mentioned above (*i.e.* interface matching, adherence to specifications, etc).

We believe that cross-cutting issues are a significant obstacle in the development of high-level mechanisms for architectural design and specification. In particular, current techniques rely on functional interfaces which obscure too much of the underlying system behavior. Moreover, such interfaces tend to be static, making it difficult to model today's dynamic systems. We believe that the techniques described in this thesis make a significant contribution towards flexible architectural interfaces. In particular, the notion of a modularity preserving meta-architecture affords the protection of current abstraction boundaries while allowing the graceful specification of cross-cutting system features. Nonetheless, significant work remains to be done with respect to analysis and verification tools. We view such tools as an evolutionary extension of the formalisms we have described in this work.

Bibliography

[1] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for unix development. In *USENIX 1986 Summer Conference Proceedings*, June 1986.

[2] G. Agha. *Actors: A Model of Concurrent Computation.* MIT Press, 1986.

[3] G. Agha, S. Frølund, W. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and modularity mechanisms for concurrent computing. *IEEE Parallel and Distributed Technology*, May 1993.

[4] G. Agha, S. Frølund, R. Panwar, and D. C. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *Dependable Computing for Critical Applications III, IFIP Transactions*, volume VIII of *Dependable Computing and Fault-Tolerant Systems.* Elsevier Science Publisher, 1993.

[5] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[6] R. Allen. Formalism and informalism in software architectural style: a case study. In *Proceedings of the First International Workshop on Architectures for Software Systems*, April 1995.

[7] R. Allen and D. Garlan. Formalizing architectural connection. In *International Conference on Software Engineering (ICSE '94)*, pages 71–80. IEEE Computer Society, 1994.

[8] R. Allen and D. Garlan. A case study in architectural modeling: The AEGIS system. In *Proceedings of Eighth International Conference on Software Specification and Design (IWSSD-8)*, March 1996.

[9] R. Allen and D. Garlan. The wright architectural specification language. Technical report, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213, September 1996. Technical Report CMU-CS-96-TBD.

[10] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[11] M. Astley. Online event-based visualization for distributed systems. Master's thesis, University of Illinois at Urbana-Champaign, May 1996.

[12] R. Barga and C. Pu. A practical and modular method to implement extended transaction models. In *Proceedings of the 21st Very Large Data Base Conference*, Zurich, Switzerland, 1995.

[13] R. Barga and C. Pu. Reflection on a legacy transaction processing monitor. In *Proceedings Reflection '96*, San Francisco, CA, USA, April 1996.

[14] D. Batory. Intelligent components and software generators. Technical report, University of Texas at Austin, Austin, TX, February 1997. Technical Report 97-06, Department of Computer Science.

[15] D. Batory, S. Dasari, B. Geraci, V. Singhal, M. Sirkin, and J. Thomas. The GenVoca model of software-system generation. *IEEE Software*, September 1994.

[16] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[17] D. I. Bevan. Distributed garbage collection using reference counting. In *PARLE: Parallel Architectures and Languages Europe (Volume 2)*. 1987. Lecture Notes in Computer Science 259.

[18] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, *Distributed Systems*, chapter 8, pages 199–216. ACM Press, New York, NY, 1994.

[19] R. Campbell, N. Islam, D. Raila, and P. Madany. Designing and Implementing Choices: An Object-Oriented System in C++. *Communications of the ACM*, pages 117–126, September 1993.

[20] J. L. Eppinger, L. B. Mummert, and A. Z. Spector, editors. *CAMELOT AND AVALON: A Distributed Transaction Facility*. Morgan Kaufmann Publishers, Inc., 1991.

[21] J. Ferber and J.-P. Briot. Design of a concurrent language for distributed artificial intelligence. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, volume 2, pages 755–762. Institute for New Generation Computer Technology, 1988.

[22] S. Frølund. Inheritance of synchronization constraints in concurrent object-oriented programming languages. In *Proceedings of ECOOP 1992*. Springer Verlag, 1992. LNCS 615.

[23] S. Frølund. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization*. MIT Press, 1996.

[24] D. Garlan, R. T. Monroe, and D. Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[25] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–146. ACM Press, New York, NY, 1994.

[26] C. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[27] C. Hofmeister, J. Atlee, and J. Purtilo. Writing distributed programs in Polylith. Technical report, Univeristy of Maryland, December 1990. Tech Report CS-TR-2575.

[28] C. R. Hofmeister and J. M. Purtilo. Dynamic reconfiguration of distributed programs. In *Proceedings 11th International Conference on Distributed Computing Systems*, pages 560–571, 1991. Also appears as Tech Report CS-TR-3119, CS Department, University of Maryland.

[29] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–75, January 1991.

[30] V. Issarny and C. Bidan. Aster: A Corba-based software interconnection system supporting distributed system customization. In *Proceedings International Conference on Configurable Distributed Systems*, Annapolis, MD, May 1996.

[31] V. Issarny and C. Bidan. Aster: A framework for sound customization of distributed runtime systems. In *Proceedings of the International Conference on Distributed Computing Systems*, Hong Kong, May 1996.

[32] Javasoft. Java Remote Method Invocation. Available at http://java.sun.com:80/products/jdk/rmi.

[33] G. Kiczales. Towards a new model of abstraction in software engineering. In *Proceedings International Workshop on Reflection and Meta-Level Architecture*, Tama-City, Tokyo, November 1992.

[34] R. B. Kieburtz and A. Silberschatz. Comments on "communicating sequential processes". *ACM Transactions on Programming Languages and Systems*, 1(2), October 1979.

[35] W. Kim. *THAL: An Actor System for Efficient and Scalable Concurrent Computing.* PhD thesis, University of Illinois at Urbana-Champaign, May 1997.

[36] D. C. Luckham. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. In *DIMACS Partial Order Methods Workshop IV*, Princeton University, July 1996.

[37] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995. Special Issue on Software Architecture.

[38] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[39] H. Masuhara, S. Matsuoka, and A. Yonezawa. Implementing parallel language constructs using a reflective object-oriented language. In *Proceeding Reflection '96*, San Francisco, CA, USA, April 1996.

[40] J. Meseguer. *Research Directions in Concurrent Object-Oriented Programming*, chapter A Logical Theory of Concurrent Objects and Its Realization in the Maude Language, pages 314–390. MIT Press, Cambridge, Mass., 1993.

[41] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *Proceedings Seventh International Conference on Concurrency Theory (Concur '96)*, Pisa, Italy, August 1996.

[42] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[43] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1–40, September 1992.

[44] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, ii. *Information and Computation*, 100(1):41–77, September 1992.

[45] Object Management Group. CORBA services: Common object services specification version 2. Technical report, Object Management Group, June 1997. Available at http://www.omg.org/corba.

[46] Open Systems Lab. The actor foundry: A java-based actor programming environment. Available for download at http://www-osl.cs.uiuc.edu/foundry.

[47] J. M. Purtilo. The Polylith software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, 1994.

[48] S. Ren and G. A. Agha. *RTsynchronizers*: Language support for real-time specifications in distributed systems. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, June 1995.

[49] S. Ren, G. A. Agha, and M. Saito. A modular approach for programming distributed real-time systems. *Journal of Parallel and Distributed Computing*, 36:4–12, 1996.

[50] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, April 1995.

[51] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, New Jersey, 1996.

[52] S. Smith and C. Talcott. Modular reasoning for actor specification diagrams. In *Proceedings Formal Methods in Object-Oriented Distributed Systems (FMOODS '99)*, Florence, Italy, 1999. Kluwer Academic Publishers.

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

[54] D. C. Sturman and G. Agha. A protocol description language for customizing failure semantics. In *Proceedings of the 13th Symposium on Reliable Distributed Systems.* IEEE Computer Society Press, October 1994.

[55] C. Talcott. An actor rewriting theory. In *Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, 1996.

[56] C. Talcott. 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.

[57] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Commun. ACM*, 33(12), Dec. 1990.

[58] The Java Team. RMI specification. Available at http://www.javasoft.com.

[59] C. Tomlinson, P. Cannata, G. Meredith, and D. Woelk. The extensible services switch in Carnot. *IEEE Parallel and Distributed Technology*, 1(2):16–20, May 1993.

[60] C. Tomlinson and V. Singh. Inheritance and Synchronization with Enabled-Sets. In *OOPSLA Proceedings*, 1989.

[61] R. van Renesse, K. P. Birman, R. Friedman, M. Hayden, and D. A. Karr. A framework for protocol composition in horus. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, August 1995.

[62] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In A. J. N. J. W. de Bakker and P. C. Treleaven, editors, *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE). Volume II: Parallel Languages*, volume 259 of *LNCS*, pages 432–443, Eindhoven, The Netherlands, June 1987. Springer Verlag.

[63] A. Yonezawa, editor. *ABCL An Object-Oriented Concurrent System*, chapter Reflection in an Object-Oriented Concurrent Language, pages 45–70. MIT Press, Cambridge, Mass., 1990.

Vita

Mark Christopher Astley was born on November 16, 1970 in Phoenix, Arizona. He attended primary and secondary school in Colorado Springs, Colorado. He graduated from Liberty High School in 1989.

In the fall of 1989, Mark enrolled in the Computer Science program at the University of Alaska - Fairbanks. While at the University of Alaska, Mark pursued a dual degree Honors curriculum in Computer Science and Mathematics. He also participated in a Department of Energy summer research program during the summers of 1991 and 1992 at the Jet Propulsion Laboratory in Pasadena, California; and was one of the first student researchers at the Arctic Region Supercomputing Center at the University of Alaska - Fairbanks. Mark graduated Magna Cum Laude with Honors and received a Bachelor of Science in Computer Science, and a Bachelor of Science in Mathematics, in May 1993.

In the fall of 1993, Mark enrolled in the Ph.D. program in the Department of Computer Science at the University of Illinois at Urbana-Champaign. While at the University of Illinois, Mark served as a Research Assistant in the Solid Modeling Lab under the direction of Dr. Yong Se Kim, at the United States Army's Construction Engineering Research Lab (USA-CERL) under the direction of Dr. Helena Mitasova, and in the Open Systems Lab under the direction of Dr. Gul Agha. In May, 1996 Mark received a Master of Science degree from the Department of Computer Science. Since the receipt of his Masters, Mark has continued his work in the Open Systems Lab, and has also served as a Teaching Assistant in the Department of Computer Science. During the summer of 1997, Mark served as a Summer Intern at IBM's T. J. Watson Research Center in Hawthorne, New York.

Upon completion of his Ph.D., Mark will be employed as a Research Staff Member at IBM's T. J. Watson Research Center in Hawthorne, New York.