interActors: A Model for Supporting Complex Communication in Concurrent Systems

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Hongxing Geng

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

In concurrent systems, such as multi-core computers, parallel systems, cloud computing systems, and systems involving mobile devices, processes interact with each other. Protocols for interactions among processes are increasingly complex and diverse, which is in part responsible for making programming of concurrent systems difficult. Particularly, in a concurrent program, the code for communication protocols often intermixes with the code for its functional behaviors, compromising modularity and reusability.

There is a growing body of work on separating communication concerns of processes from their functional concerns. Although they achieve some degree of separation, they have some disadvantages. For example, the number of communication participants is fixed in some approaches, or in other approaches, communication mechanisms, such as establishing the initial rendezvous for communication participants, are left to the processes. In other words, existing approaches either offer static protocols that cannot handle dynamically evolving number of participants in interactions, or offer complex initialization steps that are left mixed with functional concerns.

I propose *interActors*, a model for supporting complex communications in concurrent systems. I treat a communication as a *first-class* object which consists of *outlets*, through which processes can connect to it, and *handlers*, which are responsible for handling communication logics. Outlets establish a boundary between communications and processes in an application. New outlets can be created if necessary, to handle dynamically changed communication patterns at run-time. We say communications are *self-driven* because they have outlets and handlers that are active and therefore they can move interactions forward. More complex communications can be constructed by composing simpler communications.

Operational semantics and compositional semantics are developed by extending the Actor model of concurrency with support for complex communications. A prototype implementation is developed using Scala and Akka actor library. With the intention of restricting arbitrarily complex code in communications, I developed Communication Specification Language (CSL) which excludes *loops* from communications and only allows a small set of statements and expressions. *interActors* are evaluated using case studies and comparison with *Reo*, a leading coordination model and language. The evaluation shows that *interActors* offer advantages in terms of programmability, reusability, and modularity.

Acknowledgements

This work would not have been possible without the support of many people. Many thanks to my adviser, Prof. Nadeem Jamali, who consistently encouraged and inspired me and helped make some sense of the confusion. Also thanks to my committee members, Prof. Francis Bui, Prof. Mark Keil, and Prof. Chanchal Roy who offered guidance and support. I would like to thank Prof. Marjan Sirjani for her constructive comments on this work.

I would like to thank Athabasca University for giving me an opportunity to pursue my Ph.D. degree and for supporting me financially and emotionally. My colleagues always showed interest in my research, encouraged me, and showed compassion.

Many thanks go to the office staff of the Department of Computer Science. They provide a very supportive working environment and selfishless assistance to every graduate student. I would particularly like to thank Gwen Lancaster, who has reminded me of my progress reports and my registration deadlines, and has completed lots of paperwork for me.

And finally, thanks to my family – my wife Christine, my son Eric, and my daughter Emily – who endured this long process with me and always offered support and love.

This thesis is dedicated to my family.

Contents

List of Figures

List of Abbreviations

ARC	Actors, Roles and Coordinators
BPEL	Business Process Execution Language
BSPL	Blindingly Simple Protocol Language
COP	Communication-Oriented Programming
CSL	Communication Specification Language
IDL	Interface Description Language
IWIM	Idealized Workers Idealized Manager
LoST	Local State Transfer
MOM2M	Multi-Origin Many-to-Many
MOM2M2	Multi-Origin Many-to-Many (version 2)
MOM2M-2W	Two-way Multi-Origin Many-to-Many
MPI	Message Passing Interface
MPST	Multiparty Session Types
PVs	Positions and Velocities
RCF	Reflective Communication Framework
RMI	Remote Method Invocation
RPC	Remote Procedure Calls
SOA	Service-Oriented Architecture
SOM2M	Single-Origin Many-to-Many

Chapter 1

Introduction

Programming concurrent systems is difficult in part because they often involve complex and diverse interactions among computational processes. Consider, for instance, a crowd-sensed restaurant recommendation service based on data collected from diners' smartphones. A long-lived service of this type would need to group devices based on their geographic locations, solicit data from their devices, aggregate data from those at the same restaurant, etc. This requires supporting a dynamically evolving set of participants. Another thing this requires is aggregation mechanisms for transforming groups of messages into meaningful aggregates, and for deciding when sufficient information has been collected to come to useful conclusions. Given the growing need for such communications, the increasing complexity of communications, and the changing communication patterns, I argue that communications should be supported separately in a communication layer rather than handled by computational processes.

Although interactions have long been recognized as being important for open systems [81], interaction protocols are often treated as a concept secondary to computations. As a result, code for interaction protocols are often mixed with functional code, and computations use low-level primitives – locks, semaphores, or monitors in shared-variable programming paradigms or message passing, remote procedure calls, or rendezvous in distributed systems – to interact with each other.

Concurrency models, such as process calculi [38,58,59] and the Actor model [2], improve programmability of concurrent systems by abstracting away some lower-level concerns and offering a higher-level programming environment. However, as [11] convincingly argues, these models are *action-based* models of concurrency and inevitably mix interaction protocols with functional code because they still treat interaction as a secondary concept. The inadequacy of coordination support in the Actor model is discussed in [73]; similar concerns apply to other concurrency models.

There is a growing body of work on separating communication concerns from functional concerns of computational processes by promoting interaction to first-class concepts. Although they achieve some degree of separation, they have some disadvantages. For example, the number of communication participants is fixed in some approaches, and in other approaches, communication setup mechanisms, such as establishing the initial rendezvous for communication participants, are left to the computations. That is, existing approaches either offer *static* protocols that cannot handle dynamically evolving number of participants in interactions, or offer complex initialization steps that are left mixed with functional concerns. Reo [10] and BSPL (Blindingly

Simple Protocol Language) [71] are examples. In both, interaction artifacts can be constructed separately from computations. Computations can then be connected to these artifacts in order to interact with each other. Although these approaches offer powerful mechanisms for constraining flow of communication, they do not separate setting up the initial *rendezvous* for the computations participating in communications, leaving it to the computations themselves. Another deficiency is that Reo (for instance) implicitly requires that the participants should be known *a priori*. In both Reo and BSPL, once a set of processes has been linked together by the connection infrastructure, the protocols sit statically over the course of an interaction, and the processes themselves drive the communication.

In this thesis, I present *interActors* – a model for supporting complex communications in concurrent systems. By treating communications as first-class concepts, which can be created dynamically, passed to other computations, and returned as a value, communication concerns are handled separately from functional concerns. More importantly, communications encapsulate interaction protocols that describe control flows and active objects. Once launched, communications themselves can move interactions forward rather than require computations to drive them forward. In this sense, communications are *self-driven*.

1.1 Motivation

Here, I present two examples to motivate this research by showing the inadequacy of existing approaches to separating communication concerns.

1.1.1 Group Decision Making

Consider a group of employees who wish to have their shared concerns grouped together and communicated with their managers. One approach could be for one employee – the *communication initiator* – to compose a draft message and then inform the rest of the employees – the *participants* – to vote on the message. The draft message could be sent to the managers – the *recipients* – or discarded based on a *policy* that is agreed to by the sending parties. Because there is only one initiator and the message is sent to one or more recipients on behalf of a group of senders, we call this a Single-Origin Many-to-Many (SOM2M) communication [34].

Group decisions can be made in different ways, such as by voting, by authority, or by negotiation [53]. Here are two possible examples of policies:

1. *majority policy* – a majority of participants votes "yes."

2. *authorized policy* – authorized participants (maybe one or more) vote "yes" on the draft.

The first policy says that if more than half the participants agree on the draft message, the message will be sent; otherwise, it will be discarded. The second policy says that if some special participants (a subset of the entire group) agree on the draft, the message will be sent; the votes of other participants do not matter.

The votes can be gathered either by the initiator or by one of the participants. For the latter case, the initiator would inform all participants about whom they should send their votes to. Let us say that the communication's initiator is the vote collector. On arrival of a vote, the initiator (also the collector) first determines whether a decision condition has been met for the chosen policy. If it is, the initiator stops receiving votes and checks whether the draft should be sent to the recipients or should be discarded; otherwise, the initiator continues to wait for more votes. For instance, suppose the group chooses the majority policy and there are 7 participants. In this scenario, the message can be sent if 4 of the participants vote "yes." Once the initiator has received 4 "yes" votes, it stops receiving votes because it can make a *sending* decision based on the votes received so far. Similarly, the initiator can also stop receiving responses when it has received 4 "no" votes because it can make the *discarding* decision when the 4^{th} "no" vote is received.

The solution presented above embeds the communication protocol in the initiator's code. Existing approaches for separating interactions from computations do not offer elegant solutions to this problem. For example, one disadvantage of existing solutions is that as a new decision-making policy (e.g., authorized policy) is installed, significant changes have to be made on existing code. My solution offers much fewer changes on existing code by enabling communications to be reused.

1.1.2 US Presidential Election

Consider a US presidential election which involves four levels: polling station, county, state, and nation. At polling station and county levels, aggregation of votes occurs and the totals for each candidate are reported to higher levels. At the state level, reported county totals are aggregated and the states electoral college votes are awarded to the winner. At the national level, state electoral college votes are aggregated until the total number of electoral college votes for one of the candidates reaches 270, at which time the election result is announced.

For this problem, possible approaches using existing models do not scale in terms of the number of lines of code. For example, Reo – a leading interaction model – offers a solution for this problem, which requires construction of an interaction artifact called a *connector*.[1] The simplest connectors are called *channels* and more complex connectors can be built by composing simpler connectors using Reo primitives. In order to communicate with each other, computations connect to connectors. For this example, computations are voters. Interaction code for this problem using Reo would involve creating of channels, constructing of a connector from these channels using Reo primitives, and connecting of each voter to the connector. That is, the number of lines of code in Reo is in proportion to the number of eligible voters. Considering that there are about 200 million eligible voters, 113,000 polling stations, and 3,144 counties in the US, the effort involved in building such a connector and connecting each voter to it is significant. Even worse, once a connector is built, it is fixed, which means it does not accommodate the uncertainty in the nature of this problem because the voter turnout is not known before the election. There is nothing inherent about this

[1] Detailed discussions of Reo can be found in Chapter 2

problem that requires making the interactions with voters so rigid. Also, there is enough in common between interactions at the various levels that the structure of the interactions should only have to be programmed once and then parametrized for customization.

1.2 Approach

My approach for separating communication and computation concerns:

- Moves the driving force of interactions from computational processes to communications, relieves computational processes from communication tasks, and enables interactions to drive themselves
- Enables the setting up of rendezvous between communicating processes at run-time, treating it as a communication concern
- Enables creation of libraries of novel types of communications, which can be put together to create composite communications, launched, and used by applications at run-time.

1.3 Contributions

The contributions of this work are as follow:

- Built the *interActors* model on the Actor model, present its operational semantics and compositional semantics, and studied the observational equivalence of communications. *interActors* separate first-class communications from computations. Communications can be composed to build more complex communications. Also, communications are self-driven.
- Implemented a proof-of-concept prototype of *interActors*.
- Developed Communication Specification Language (CSL), a language for writing executable specification for communications.
- Evaluated *interActors* through case studies and comparison with Reo – a leading interaction model.

1.4 Outline

The rest of this thesis is organized as follows: Chapter 2 discusses related work. Chapter 3 introduces the concepts of communications and their compositions. Semantics of *interActors* including operational semantics and compositional semantics are presented in Chapter 4. A prototype implementation is described in Chapter 5. Chapter 6 presents the Communication Specification Language. *interActors* are evaluated with respect to programmability, modurality, and reusability in Chapter 7. Finally, Chapter 8 concludes this thesis.

Chapter 2

Related Work

There is a growing body of work focusing on separating computations' interaction concerns from their functional concerns. This chapter discusses related works in middleware, software architecture, and coordination models and languages. Section 2.1 discusses middleware. Section 2.2 presents related works in software architecture. Coordination models and languages are described in Section 2.3. Reflection-based approaches, which are coordination models, are discussed in Section 2.4.

2.1 Middleware

I classify middleware into two categories based on their communication primitives. Low-level middleware expose low-level communication primitives to programmers. In contrast, high-level middleware provide higher-level communication primitives.

Low-level Middleware

A number of middleware systems try to separate coordination from computation. They do so by providing low-level primitives to address specific narrow deficiencies. For example, CORBA [78] is designed to facilitate communications between different operating systems and programming languages. It provides an Interface Description Language (IDL) to describe object interfaces used to communicate with the external world. MPI (Message Passing Interface) [80] offers language-independent communication protocols particularly designed for parallel computing. MPI programmers can use communication primitives for both point-to-point (e.g., *MPI_Send* and *MPI_Recv*) and group communication (e.g., *MPI_Bcast* and *MPI_Scatter*).

Java Remote Method Invocation (Java RMI) [65] is the object-oriented equivalence of Remote Procedure Calls (RPC). An application using Java RMI has three components: an interface that declares remote methods, a client class that calls the remote methods, and a server class that implements those remote methods. To interact with a server, a client makes a call to a remote method, which is executed by the server and a response is returned. Separation of concerns is realized through the declaration interface, as IDL interfaces in CORBA.

These low-level middleware separate interfaces from their implementations. However, communication protocols are mixed with functional code.

High-level Middleware

Publish/Subscribe communication middleware implements the publish-subscribe messaging pattern [29]. It decouples publishers and subscribers in terms of space and time. Space decoupling means that the communicating parties do not need to know each other; time decoupling means the communicating parties can interact with each other asynchronously. Subscribers can selectively receive published messages by publishers. In other words, subscribers only receive a subset of the total published messages. This is realized through either *topic* or *content*. In *topic-based* systems, subscribers only receive messages published to their subscribed topics; in *content-based* systems, subscribers receive messages matching constraints defined by them. There are a growing body of work on publish/subscribe systems recently. It is used in various applications, such as wireless network [40], peer-to-peer systems [69], and location-awareness system [22].

Stigmergy is another communication scheme which decouples communicating parties. The term stigmergy [74] was coined by the French biologist Pierre-Paul Grassé to refer to termite behavior. Loosely speaking, stigmergy is a type of *indirect* interaction, in which termites can communicate with one another by modifying environment. By *indirect*, we mean senders do not acquaint with receivers. Strictly speaking, stigmergy interaction can be split into two communications: one is when a termite interacts with environment by leaving pheromones; and the other is when other termites interact with the environment by extracting information from pheromones left by others. Stigmergic communication has been discussed in the context of multiagent systems [54]. Further, researchers and practitioners are attempting to take advantage of stigmergy so as to serve human life altogether. For example, [28] is an attempt to promote a social site through exploiting stigmergy to identify and analyze online user behaviors. Human-human stigmergy is studied in [76] in the context of multi-agent systems.

The difference between stigmergy and publish/subscribe is that in the former, receivers have to actively *pull* information from the environment; whereas, in the latter, systems actively *push* information to receivers (i.e., the subscribers). High-level middleware, such as publish/subscribe and stigmergy, provide separation of communication from computations; however, they only focus on particular communication patterns.

2.2 Software Architecture

From the software architecture point of view, a software system is a composition of computational *components* responsible for functional concerns and *connectors* stipulating interactions between components. By dividing a software system into components and connectors, separation of concerns is achieved. That is, components focus on computations and connectors are responsible for interactions between components.

Software Connectors

The concept of software connectors was first proposed by Mary Shaw in [70]. The paper defined connectors as "mediate interactions between components." In this sense, an application consists of a collection of components and a group of collectors. A comprehensive classification on connectors is discussed in [55], in which a connector is defined in terms of channels. A connector comprises one or more channels. Each channel provides mechanisms for transferring data and control. In addition to mechanisms, [44] argues that connectors also require agreements because components participating in communications must share some common assumptions in order to communicate with each other without confusion. For instance, assumptions can be the type of data transferred, the syntax and semantics of the messages exchanged, and data encoding conventions, etc. When considering system run-time, a connector must be instantiated. Hence, connectors should also have types, instances, and states [44].

Although software connectors separate communications from computations to some extent, components are still responsible for coordinating the control flow. In other words, a component not only performs computation but also initiates method calls and manages their returns, leading to mix computation with control. [48] proposed *exogenous* connectors. Exogenous connectors encapsulate control and data flow between connected components, enabling control to originate from connectors rather than from components. A software component models for exogenous connectors has been proposed in [46].

Collective Interfaces

Although there is a body of work on software connectors, there is no approach specifically addressing collective communication for distributed components. Collective interfaces [14] address collective communications for distributed components through interfaces. Collective interfaces based on the Fractal component model [17], in which components communicate with each other through their interfaces. However, the model does not support collective communications. Collective interfaces extend the Fractal model by allowing one-to-many and many-to-one communications. Two collective interfaces are proposed, namely, *multicast* interface and *gathercast* interface. A multicast interface enables parallel invocations by converting a single invocation to a list of invocations. That is, through it, a message is sent to multiple components, and each of them can process the message in parallel. A multicast interface enables a sender to multicast a message instead of sending multiple individual messages. A gathercast interface enables many-to-one communications by transforming a list of invocations from different sources to a single invocation. That is, a gathercast interface synchronizes incoming messages and sends one aggregated message to a component.

Web Service Composition

Web service composition [6] adopts the concept of Service-Oriented Architecture (SOA). Applications consist of a number of web services. The goal of web service composition is to compose a number of web services

into a work flow so that the web services are composed into an executable business process [62]. Web services are combined in two ways: orchestration and choreography. In orchestration, a number of web services are composed through a central web service, which orchestrates invocations of different operations on different participating web services. The participating web services have no idea whether they are in a composition process. The control logic for an invocation sequence is described by Business Process Execution Language (BPEL) [8], which is a declarative language that supports web service orchestration and separates web services and interactions between them. Choreography, in contrast, does not require a central process. Each web service knows the business process and therefore understands in what order operations should be executed.

By comparing orchestration and choreography, we can see choreography mixes computations with communications to some degree; whereas, orchestration, through a dedicated process, separates interactions from computations successfully.

2.3 Coordination Models and Languages

The increasing need for separating coordination concerns from computation concerns gave birth to research in coordination models and languages.

Linda

Linda [33] is thought to be the first coordination language to separate coordination concerns from functional concerns. Computations interact with each other only through shared objects called *tuples*, which reside in a shared place called *tuple space*. Synchronization is achieved through a blocking read primitive.

Linda defines three communication primitives: `out`, `in`, and `rd`. `out` inserts a tuple with a tag without targeting a specific recipient into the tuple space. A computation can ask to read a tuple using either `in` or `rd` by providing a tag. If there is tuple matching with the tag, it is read; otherwise, the reader waits. `rd` reads the tuple but leaves it in the tuple space; `in` also removes it. Although Linda is said to offer *time-decoupling* (computations can communicate asynchronously) and *space-decoupling* (computations do not need to know each other), it mixes communication protocols with functional behaviours because the communication primitives must be incorporated within computations.

Programmable Environments

A number of works [18] [61] [82] share the idea of programmable environments. The general idea is to make the host environment of concurrent computations reactive by extending the environment with *behaviors*. Among them, we choose *tuple center* [61] as an example to describe how they work.

A tuple center [61] is a coordination medium, which extends the standard tuple space with a behavior. A behavior associates a communication event (i.e., a call to a communication primitive) to a set of computational activities called a *reaction*. A behavior is specified in terms of a *reaction specification language* named

`ReSpecT` [25], which is based on first-order logic and is a language used for programming tuple centers. When a communication event occurs, it may lead to execution of a *reaction*. A reaction may manipulate tuples in the tuple center or trigger other reactions in a chain. Because reactions can be added or removed dynamically, the behaviors of a tuple center can be changed on the fly.

Idealized Workers Idealized Manager

Idealized Workers Idealized Manager (IWIM) [9] supports coordination through a dedicated *manager* that coordinates the interactions among multiple worker processes, which is similar to web service orchestration. Communications between workers are through channels, each of which has one input end and one output end. Workers communicate with each other through writing to and reading from their connected channel ends. However, workers and channels are oblivious to each other, and therefore they do not know how to make connections. The task of a manager is to create and destroy channel connections between worker processes so that those processes can communicate with each other; whereas, worker processes are responsible for computation. In other words, a manager constructs a communication network using channel manipulation primitives and connects channels to worker processes. Hence, in IWIM, the separation of coordination from computation is achieved through a manager. However, because workers communicate with each other through channels, when a large number of workers are present, the workload of a manager, which is responsible for creating channels and connecting them to processes, could be tedious and overwhelming. That is, IWIM is not scalable.

Reo

Reo [10], a leading approach in coordination models and languages, offers a channel-based dataflow-oriented coordination model. Computations can interact with each other only through *connectors*. The most primitive and simplest connectors are called *channels* and more complex connectors can be constructed by composing simpler connectors. A channel has two ends, namely, the *source* end, to which a data item is stored through a *write* operation, and the *sink* end, from which a data item is read via a *read* operation. A channel is created using a *create* operation. Reo assumes arbitrary number of channel types and thus allows channels types to have user-defined semantics. A small set of commonly-used channels with simple behaviors is pre-defined, which can be used for synchronization, asynchronization, filtering, and transformation purposes. These channels are intended to be sufficient for constructing connectors with more complex behaviors through *channel composition*, which is realized through *join* operations. For example, an *alternator* allows a computation to alternatively receive messages from a set of computations, a *sequencer* enables messages to be taken in a defined sequential order, and a *barrier* implements barrier synchronization.

Reo has an ingenious way of orchestrating communications by using channels with special features to direct messages flowing through a connector. However, some coordinator/initiator process has to construct required connectors by performing operations such as *create*, *connect*, and *join*. Setup for a complex system with a large

number of components requires creating a large number of channels, composing them to create connectors, and connecting channel ends to components, Not to mention that creating special purpose channels and putting them together in a connector too is non-trivial for more complex interactions. If communication requirements of a computation were to change at run-time, components would have to be disconnected, a new connector would need to be created, components would have to be connected again, and presumably an unwieldy suspension and resumption of the interaction would be required. Related to this drawback is also the implicit requirement that the participants connected by a connector are known in advance and do not change at run-time. There is no obvious clean way for accommodating such a change.

Session Types

Session types [26] can be seen as a language to define communication protocols for communicating parties, therefore, it can be considered as coordination languages. A *session* is a sequence of interactions between two parties. Session types regulate the types of messages received or sent by a party and thus specify communication protocols between two parties. Using session types to define communication protocols usually spreads the definition among communicating parties. For example, to describe a communication protocol between a buyer and a seller, there are two separate definitions composing the entire protocol: a protocol definition for the buyer and a protocol definition for the seller. These two definitions are dual to each other. Multiparty Session Types (MPST) [39] extends session types from two parties to multiple parties. Communication protocols using MPST are again dispersed among those communicating parties. That is, a protocol definition must be provided for each involved party.

Blindingly Simple Protocol Language

There are also declarative approaches to build interaction protocols. A representative example of this is the Blindingly Simple Protocol Language (BSPL) [71]. BSPL describes interaction protocols among computations declaratively. Protocols are treated as first class entities and can be composed to form more complex protocols.

Each BSPL protocol has a unique name and includes two or more *roles* instantiated by interaction computations, one or more parameters, and one or more references to constituent protocols or message schemas. One parameter or some combination of parameters is defined as a *key*, which is the unique communication identity when a protocol is instantiated. Furthermore, when using a protocol, it may require parameters input from outside of the protocol and it can output values to the external world. If a protocol includes parameters requiring input from the outside, the protocol must be used in combination with other protocols which can provide values for those inputs. Message schemas described in a protocol define messages that can be exchanged by two roles. Each message schema not only regulates which role is the sender and which is the recipient, but also stipulates what information can be exchanged through defined parameters, among which at least one indicates the unique communication identity. Protocol composition is achieved through references to constituent protocols.

BSPL is information-oriented, meaning that it makes information exchange explicit and it can impose the required flow of information. It treats interaction protocols as first-class concepts, where simpler protocols can be composed to form more complex ones. Although these are important features, it suffers from many of the drawbacks of Reo. BSPL implicitly constrains the number of participants in a communication, and the types of protocols to those involving fixed numbers of participants. A realization mechanism is presented in Local State Transfer (LoST) [72], which is a declarative approach for enacting communication protocols written in BSPL.

Reactor Model

Reactors [66] are concurrent computations encapsulating channels and event streams. Events are messages. Every channel belongs to a single reactor and has an associated event streams. Inter-reactor communications are through channels and intra-reactor communications are through event streams. That is, a reactor can send an event to a channel, which propagates the event inside the recipient reactor. One reactor can send an event to another reactor as long as the former has the name of one of the latter's channels. In other words, reactors do not interact with each other directly but through their associated channels, where communications can be processed. Hence, the reactor model separates communication from computations. A reactor can create or terminate a channel as necessary. Channels and event streams are first class entities. As a consequence, they can be used to compose communication protocols. A protocol can be encoded as functions or as classes and can be used by reactors. When a reactor uses a protocol, the channels and events declared in the protocol are instantiated. A reactor can participate in multiple communication protocols using separate channels. A protocol instance can be used to customize the communication of one *single* reactor; however, a definition of a protocol can be reused across multiple reactors.

MapReduce

Developed by Google, MapReduce [24] is a high-level programming framework for processing and generating large-scale datasets, which applies the master-workers interaction pattern in its implementation. MapReduce defines a work flow for a master and worker processes, and computations are carried out by programmer-provided functions. A computation in MapeReduce is expressed as one of two tasks: *map* and *reduce*, which are two functions written by programmers. The MapReduce library in a user program creates multiple copies of the program. Among those newly spawned program, one is the master and the rest are workers. The master assigns either a map task or a reduce task to idle workers. Workers executing map tasks read input content and output intermediate results, which are buffered in memory and are written into local disks periodically. The master is notified with the locations of those intermediate results. Workers performing reduce tasks obtain location information from the master, read the intermediate data, and compute final results, which are returned to the master, which, in turn, returns the result to the user program. Like web

service orchestration and IWIM, MapReduce achieves separation of concerns through a dedicated master process.

Other Coordination Models and Languages

There are many other coordination models and languages developed for various domains, such as mobile ad hoc networks [50, 63, 64], service-oriented systems [1], fault-tolerance systems [52], social collaboration [49], and distributed applications [23, 83], etc.

For example, as a coordination language, *Statelets* [49] provides a unified access to social collaboration processes spanning multiple groupware tools and social networking sites. A statelet is a construct indicating a state of a process and designating coordination rules which should be fulfilled in that state. A process is a sequence of statelets which create more statelets. A statelet consists of two principle components: *condition*s and *action*s. A condition describes an anticipated situation and an action will be executed if such a condition is detected. Conditions are illustrated in the form of context queries that are constructed using binary operators of first-order logic and are evaluated using two primitive operations: `define` and `wait`. `define` merely evaluates a query expression and searches shared space for pattern instances; whereas, `wait` operation constantly evaluates a query until at least one pattern instance is found. Both operations dynamically creates a data stream which is similar to a tuple in Linda. Data streams will be used in an action. An action may either perform commands or create new statelets.

2.4 Reflection-based Approaches

A number of coordination models have also been developed using the idea of computational reflection [51]. To separate interaction concerns from functional concerns, the idea is to add new layers into the system. Interactions are handled in the new layers and computations reside in the computation layer. These approaches include *syncrhonizers* [27, 31], protocol stacks [13], Reflective Communication Framework (RCF) [36], ARC (Actors, Roles and Coordinators) [67], and ActorSpaces [19].

Synchronizers

Synchronizers [31] is a language framework for multi-object coordination, which enforces declarative synchronization constraints called *synchronizers* on a group of objects. Conceptually, a synchronizer is a special object which observes messages and restricts message dispatch for other objects in accordance with user-specified message patterns. In other words, messages matching a synchronizer's message pattern fall under the synchronizer's control. Synchronizers have a global effect and are imposed on receivers. Also, synchronizers may overlap, which means that different synchronizers can constrain the same object. Synchronizers provides two types of constraints: (*a*) *disabling* constraints block the constrained objects from handling

messages which match a given pattern, (*b*) *atomicity* constraints ensure that bundled messages to multiple objects should succeed atomically.

Built from synchronizers, scoped synchronizers [27] is a coordination model based on declarative synchronization constraints. To unravel the issue that components in Internet-scale systems cannot be trusted, scoped synchronizers restricts applications of synchronization constraints to a limited scope rather than having a global effect. Like conventional synchronizers, scoped synchronizers are declarative synchronization constraints with disabling constraints and atomicity constraints. Unlike conventional synchronizers, scoped ones enforce constraints on the sources of messages rather than on receivers.

Protocol Stack

A protocol stack [13] enables communications of computations to be customized through metaobjects. Each metaobject implements a communication protocol and can only customize communications for a single computation. A communication protocol enforced on a group of computations is implemented using the metaobjects collectively customizing each computation. A metaobject itself can be customized by a meta-metaobject so that a computation can be customized through a metaobject protocol stack. A stack can be changed on the fly by installing a new metaobject or removing an old metaobject, allowing two consecutive messages of a computation to use different protocols. In this sense, metaobjects can customize communications on a per-message base.

Despite its benefits, this approach is too rigid in that protocols need to be explicitly added on or removed from the stack in response to changing communication requirements. Furthermore, both incoming and outgoing messages share the same protocol stack, which can lead to excessive overhead. Any change in communication requirements of a computation requires removal of existing metaobjects, installation of new metaobjects, or replacement of metaobjects with new ones, leading to changes in the interfaces between metaobjects and between metaobjects and the computations they customize. Last, for messages which do not require communication protocols, they too have to pass through the entire protocol stack, incurring unnecessary computational overhead.

Reflective Communication Framework

In Reflective Communication Framework (RCF) [37], actors share a single communication protocol pool managed by a *communication manager*. Communication protocols are implemented as actors. Like metaobjects, in RCF, each actor has a corresponding meta level actor called *messenger*, which serves as the customized and transparent mail queue for its base level actor. A messenger has four mail queues: an *up* and a *down* mail queue is used for an outgoing message and an incoming message, respectively, which do not require communication services from the communication manager; otherwise, an *out* and an *in* mail queue are used. In other words, *up* and *down* mail queues serve traditional actor message; and *out* and *in* mail queues are designed to customize communications. From this sense, RCF has performance advantage over metaobjects

because in metaobjects, every message must go through the entire protocol stack; however, in RCF, if a message does not require customization, it can use *up* or *down* queue.

In addition to keeping track of communication protocols, the communication manager maintains a set of *communication message coordinator*. When a message requires communication service, the communication manager assigns it to a *communication message coordinator*, which is responsible for composing the communication protocols requested by the communication's messenger. Once its job has been done, a communication message coordinator is available to serve next message. New communication protocols can be added dynamically into the protocol pool, which makes RCF suitable for dynamic environments.

Compared to metaobjects, RCF offers a more elegant way for customizing communications. However, like metaobjects in which every message must go through the entire protocol stack, in RCF, every individual message must go through its messenger even if the message does not require communication services, which causes computation overhead. Moreover, it does not support protocol enforcement on groups of computations in the way that metaobjects do, and does not support interactions between protocols.

Actors, Roles and Coordinators

Actors-Roles-Coordinators (ARC) [67] is a coordination model composing three layers: the actor layer, the role layer, and the coordinator layer, from bottom to top. Actors are dedicated to functional behavior; roles are static abstractions for coordinated behaviors shared by a group of actors and provide localized coordination among those actors; and coordinators coordinate different roles. In other words, there are two types of coordination in ARC: intra-role coordination realized by the role layer and inter-role coordination implemented by the coordinator layer. Coordination among actors is through message manipulations that are transparent to the actors.

The actor layer is dedicated to computational behavior without knowledge of the coordination enforced by the role layer and the coordinator layer. Conversely, the coordinator layer is oblivious to the actor layer and is dedicated to inter-role coordination which focuses on coordination among roles. The role layer is in the middle and is reserved for intra-role coordination, which coordinates actors within a role.

ActorSpace

ActorSpace [19] is a communication and coordination model based on destination patterns. ActorSpace extends point-to-point asynchronous message passing in the Actor model with pattern-direct invocation, which was inspired by Linda. Through patterns, a group of receivers is defined; thereby, receivers can be anonymous to senders, decoupling receivers from senders.

An actorspace is a computationally passive container of actors which provides a scoping mechanism for pattern matching. In ActorSpace model, each actor has a list of attributes, so do actorspaces. Like actors, an actorspace has a unique name. An actorspace may contain actors and other nested actorspaces, which may be made visible or invisible in an actorspace. Patterns use attributes to define a group of receivers and

are matched against listed attributes of actors and actorspaces that are visible in the actorspace. In other words, only visible actorspaces and actors are subject to pattern-matching in ActorSpace.

Communication in ActorSpace is through two primitives: `send(pattern, message)` and `broadcast(pattern, message)`, which augment the `send` primitive in the Actor model. The destination of `send` is defined by `pattern` other than an actor name as in the Actor model. ActorSpace supplies a `broadcast` primitive which is missed in the Actor model. `send(pattern, message)` sends a message to a single actor which is non-deterministically chosen from the group of potential receivers defined by `pattern`, which is similar to IP anycast [56] insofar as communication is concerned. On the other hand, a message is sent by `broadcast(pattern, message)` will be received by all actors in the group.

Chapter 3

Communications

Without communication, each computation in a concurrent system would become an isolated island. Despite its importance, communication is typically treated as a secondary concept, and consequently, communication concerns are intermixed with functional concerns of computations. This chapter presents a different view about communications. With the intention of programming complex communication concerns for applications separately from their functional concerns, a communication is treated as a *first-class* object. By first-class, we mean communications can be created dynamically, be passed to other computations, be returned as a value, and be destroyed. Because of the characteristic of first-class, the creation and usage of a communication can be separated. That is, the creator and the user of a communication can be two different objects.

Section 3.1 introduces our approach of separating communication and computation concerns. Then, Section 3.2 presents the definition of *communication* and describe its components: *outlets* and *handlers*. Section 3.3 describes three composition rules through which more complex communications are constructed from simpler communications. Section 3.4 summarizes this chapter.

3.1 Introduction

When an electric appliance is to be used, it is plugged into a power outlet. When it is no longer needed, it is unplugged. Similarly, in order to interact with each other, computations plug into the *outlet*s of a communication. Figure 3.1 illustrates the relationship between computations and communications. In this figure, the rectangle represents a communication, which has a set of outlets; ovals are computations. We use the shape of power outlets to represent outlets and use the shape of plugs to denote the connection point of computations. In order to participate in a communication, computations have to connect to the communication's outlets through their plugs. To leave a communication, computations simply disconnect from the communication's outlets like an appliance is unplugged from a power outlet. In this way, computations are separated from communications.

Outlets establish a *boundary* between communications and computations in an application. Computations and communication are oblivious to each other, thus separating concerns of communications from

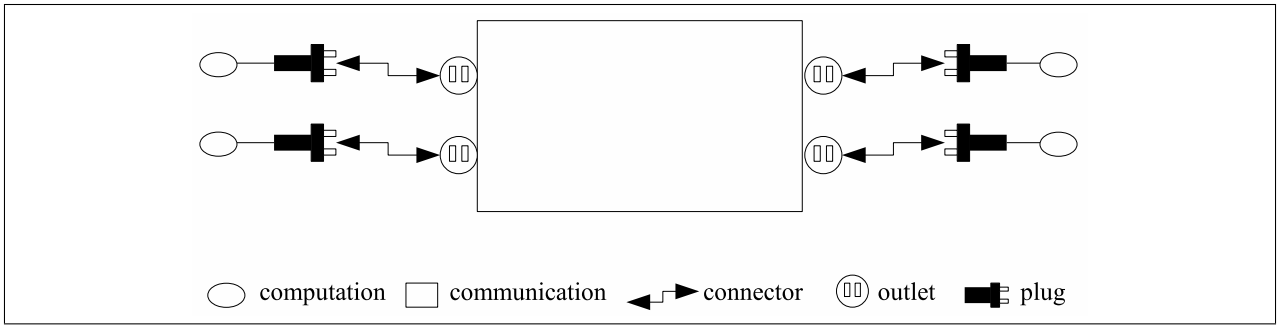

Figure 3.1: Interaction through a Communication

concerns of computations for an application. In this way, both communications and computations can evolve independently, improving reusability.

3.2 Communications

A communication includes a set of *outlets* and a set of *handlers*. Outlets can be one of two types: *input* and *output*. A communication receives messages from computations through its *input* outlets and sends messages to computations through its *output* outlets. Handlers implementing communication logic process the received messages. Our definition for *communication* is as follows:

Definition 1. *A communication has at least one input outlet, which receives messages from communicating parties, and at least one output outlet, which sends messages to communicating parties, and a set of handlers, which are responsible for handling communication logics.*

From the definition, a communication has two types of components: outlets and handlers. Outlets and handlers are active objects. Outlets interact with computations through sending and receiving messages and handlers process messages received from input outlets and send processed messages to output outlets. From this sense, a communication is a set of active objects. Input outlets are *receptionists* of a communication. That is, computations must know input outlets of a communication in order to participate in the communication. Output outlets must hold the names of computations so as to send messages to them.

Figure 3.2 illustrates a communication. In the figure, ovals are handlers, white circles are input outlets, black circles are output outlets, and the lines with arrows represent message flows; the communication encapsulates handlers. Handlers are invisible to computations. To interact with a communication, computations have to be connected to the communication's outlets: to send messages to the communication, computations are connected to its input outlets; to receive messages from the communication, computations are connected to its output outlets.

To connect to an outlet, a computation must know its name. A computation can obtain the name of an outlet through two ways: one is as the creator of a communication; another way is from its received message containing the name of an outlet. Because an outlet may be known by multiple computations, it

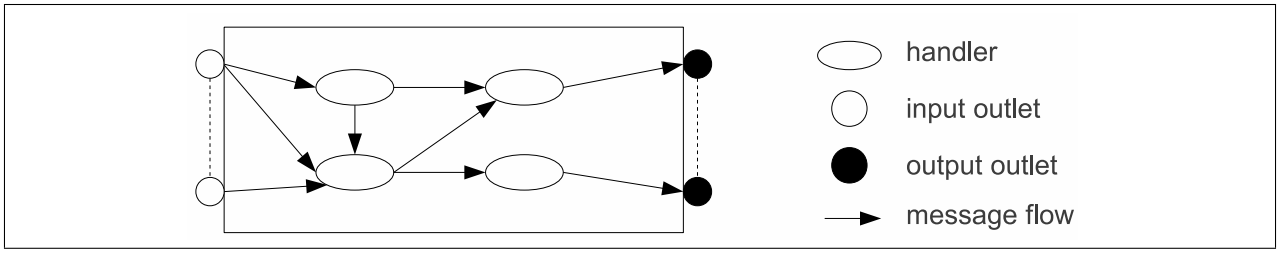

Figure 3.2: Communication

can communicate with those computations connecting to it. Likewise, a computation may participate in multiple communications at the same time as long as it knows outlets of those communications. Therefore, the relationship between computations and outlets is *many-to-many*.

3.2.1 Outlets

A communication must have outlets because otherwise, computations cannot participate in it and it will be useless. An outlet is an active object and is a receptionist of the communication. Computations only need to connect to the outlets of a communication in order to participate in it. Outlets make computations and communications oblivious to each other.

Each outlet can be one of two types: *input* or *output*. Messages can be received at a communication's input outlets and sent out from its output outlets. Each outlet has a behavior, which defines what the outlet does when receiving a message and where the processed message should be sent to. More precisely, an input outlet of a communication receives messages from computations and sends processed messages to the communication's handlers or the communication's output outlets, which, in turn, forward those messages to computations. We describe behaviors in more detail in Section 3.2.3.

3.2.2 Handlers

Handlers hold interaction logic. A handler can send messages to other handlers or outlets. A communication may not need a handler if it does not require any interaction logic. For example, messages sent to a point-to-point communication are just simply forwarded from the input outlet to the output outlet without requiring any complex logic. In other words, the set of handlers of a communication can be empty. Like outlets, handlers present a behavior, which are discussed in Section 3.2.3. Handlers can change outlets' behaviors by sending them a special message, which will be discussed in Chapter 4.

3.2.3 Behaviors

Outlets and handlers have behaviors. A behavior defines how an outlet or a handler reacts with incoming messages. Each behavior also has *targets* which defines where processed messages are sent to. The simplest behavior can be *forwarder*, which simply forwards whatever it receives to its targets. Suppose, a communi-

Behavior	Description
forwarder(msg, ts)	forwards a received message *msg* to targets *ts*.
counter(msg, ts)	counts the number of received messages so far and sends the result to targets *ts*.
timer(msg, ts, t)	temporally holds the received message *msg* for *t* milliseconds and then sends it to targets *ts*.
applier(msg, ts, f)	applies function *f* to a received message *msg* and then sends the result to targets *ts*.
filter(msg, ts, f)	filters a received message *msg* based on function *f*. If *msg* meets the requirements of *f*, it is sent to targets *ts*; otherwise, discarded.
selector(msg, ts, select)	sends the received message *msg* to a subset of targets *ts* selected by function *select*.
sequencer(msg, ts, parts)	a sequencer sends the received message *msg* to a list of participants *parts* one by one. Once all participants have responded, the received message is sent to the sequencer's targets.
aggregator(msg, ts, cond, aggr)	on the arrival of a message, the message *msg* is added into a message list; then function *cond* is applied to the message list to determine whether a certain condition is met or not. If it is, all received messages so far (stored in the message list) is aggregated using function *aggr* and the aggregated result is sent to targets *ts*.

Figure 3.3: Examples of Behaviors

cation has two outlets – one input and one output, both of them have *forwarder* behavior, and the input outlet's target is the output outlet. As to this communication, its input outlet simply forwards whatever it receives to its output outlet, which forwards received messages to computations that connect to it.

Outlets and handlers can also have more interesting behaviors. For example, an outlet can count the number of received messages, filter out unwanted messages, aggregate a number of received messages, apply functions (such as for encryption) to received messages, or temporally delay received messages by forwarding them at a future point in time.

We assume there is a library of behaviors. Programmers can freely add new behaviors into the library. Here I list commonly used behaviors in Figure 3.3 and discuss each of them below.

forwarder

The most straightforward example of a behavior is *forwarder*, which forwards a received message to its targets. For an input outlet, its targets are usually handlers or output outlets; and for an output outlet, its targets are computations.

As an example, *forwarder* can be used in a point-to-point communication, which has one input outlet and one output outlet. Both outlets have the behavior of *forwarder*. The target of the input outlet is the output outlet, whose target is a computation.

counter

A counter counts the number of received messages so far and sends the result to its targets. A counter is useful when a computation has interest to know the number of messages but not the contents of messages.

timer

A timer temporally delays received messages for pre-defined time period and then sends them to its targets. An example is Boomerang [16], which schedules email to be sent later in Gmail.

applier

An applier applies a programmer-defined function to received messages and sends the result to its targets.

filter

A filter uses a programmer-defined function to filter received messages. Only messages satisfying the requirements defined by the programmer-defined function can be sent to the filter's targets. A filter can be seen as a special case of an applier because it applies a function to messages and returns a boolean value.

selector

A selector selects a subset of recipients from its targets based on received messages using a programmer-defined function and sends received messages to those chosen targets.

sequencer

A sequencer sends received messages to a list of participants in a sequential order. In other words, it sends the received message to the first participant and waits for a response; after receiving a response, it sends the response to the second participant, and so on, until all participants have responded. Finally, the response from the last participant is sent to the targets.

aggregator

An aggregator computes an aggregation result from received messages so far when certain conditions are met, and sends the result to its targets.

3.2.4 Channel – The Primitive Communication

The primitive communication is a *channel*. Figure 3.4 shows a channel, which simply connects one input outlet (the white circle) to one output outlet (the black circle). It enables sending of asynchronous messages from a process connecting to the input outlet to another process connecting to the output outlet.

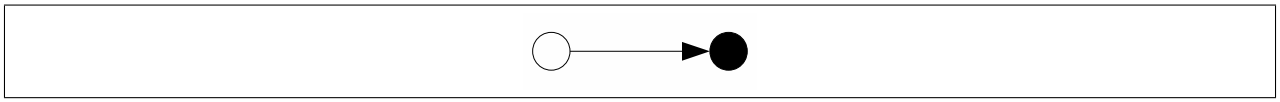

Figure 3.4: Channel

3.3 Composition

We define complex communications compositionally. In other words, what we call a complex communications is defined in terms of simpler communications using three composition rules described below. A complex communication is anything that can be constructed by beginning with channels and repeatedly applying these three rules: *input merge*, *output merge*, and *output-input merge*.

3.3.1 Composition Rules

This section describes the three composition rules: *input merge*, *output merge*, and *output-input merge*.

Input Merge

As illustrated in Figure 3.5, an *input merge* composition merges a number of communications at their input outlets. The number of communications being merged is not decided *a priori*, and is determined dynamically by the number of relevant recipients who fit some *pattern* [3]. The purpose is to enable a single sender to send messages to a number of recipients. A computation connecting to the input end of this communication can send messages, each of which is then delivered to each of the computations connected to the output outlets. The input outlet at the merging point has a behavior, which enables a message to be processed before sending forward.

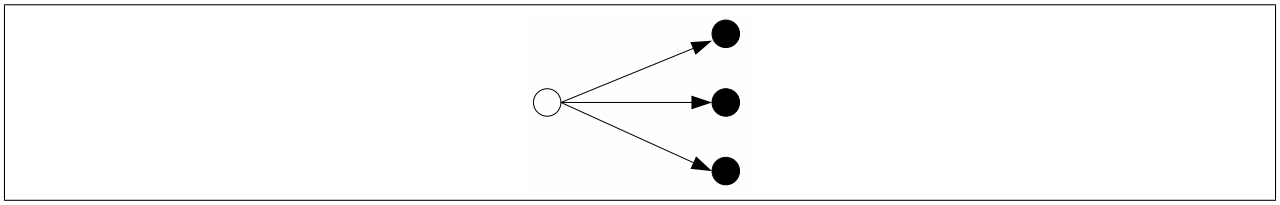

Figure 3.5: Composition: Input Merge

At the first glimpse, one may think this composition is the same as *broadcast*. They are the same when the input outlet just simply forwards received messages to output outlets. However, it can be significantly different

from *broadcast*, because the input outlet in this composition has a behavior, which enables the communication to realize functions such as counting, aggregating, scattering (e.g., MPI scatter), etc. Furthermore, a message at the input outlet may be sent to some targets but not to others, which we call *selective* sending, which is similar to Akka routing [75].

Output Merge

Figure 3.6 shows an *output merge* composition, which merges a number of communications at their output outlets. The purpose is to enable a single recipient to receive the messages sent by a number of non-deterministically determined senders in some order. The number of communications merging their output outlets is not known *a priori*.

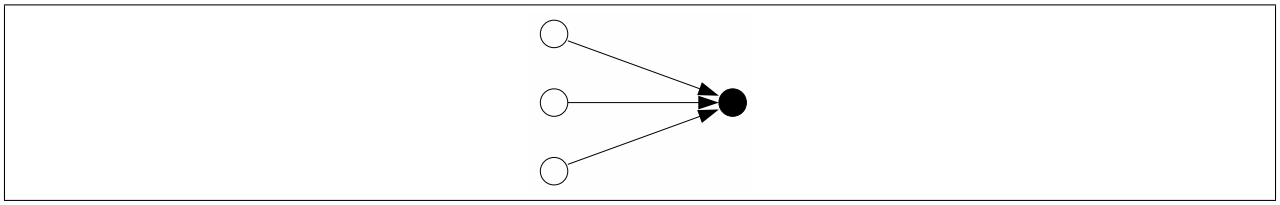

Figure 3.6: Composition: Output Merge

At the output end, the messages can be received by a computation and processed as required by the computation. Alternatively, they can be somehow aggregated. In its simplest form, the aggregation function simply spits out each message received in its original form. In more interesting forms, it can process received messages in permitted ways, both to create aggregate messages to be forwarded, and to make decisions about whether and when to forward aggregates. For example, the aggregation could impose constraints on delivery of messages from different sources in a way similar to how the channels in Reo do, except that the number of computations participating in a communication do not have to be known *a priori*. Also, the protocols can be finer-grained and rely on decisions possible only at run-time. For example, a protocol could wait for a majority of voting computations to vote "yes" or "no," counting each vote equally, or weigh the votes of different voters differently.

Output-input Merge

Output-input merge connects output outlets to input outlets based on bindings, each of which pairs one output outlet of one communication with one input outlet of another communication. Figure 3.7 shows merging one output outlet with one input outlet. The doubled arrow is the *composition glue* which shows the connection and message flow. Notice that unlike *input merge* and *output merge*, the types of two merged outlets are different: one is output and another input. Moreover, the message flow is from the output outlet to the input outlet, but not vice versa.

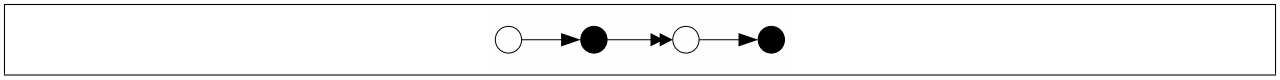

Figure 3.7: Composition: Output-input Merge

3.3.2 Examples

More complex communications can be composed from simpler communications by using the three rules repeatedly. This section gives three examples.

Example 1 Figure 3.8 shows an example using all three composition rules to compose six channels. C_1, C_2, and C_3 are composed by merging their output outlets; C_4, C_5, and C_6 are composed by merging their input outlets; these two composed communications are composed by applying the *output-input merge* rule. The final communication has three input outlets and three output outlets.

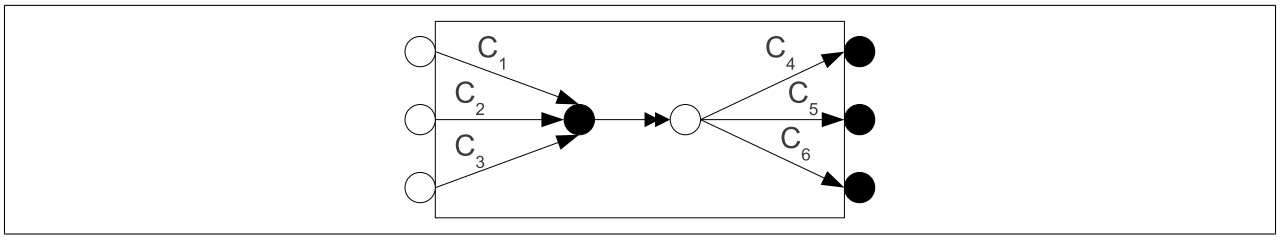

Figure 3.8: Example 1: Communication Composition

Example 2 Figure 3.9 shows an example of composing two communications – C_1 and C_2 – using the output-input merge rule. Each communication has three input outlets and three output outlets. The two smaller solid rectangles represent the two communications to be composed and the dashed rectangle represents the *composition glue* which composes them. The particular composition happens by connecting two of the output outlets of C_1 to two input outlets of C_2, and one output outlet of C_2 to one input outlet of C_1. The composed communication is represented by the biggest rectangle. To an external observer, the composition glue is invisible, as are the six outlets it connects. In other words, an external observer only sees three input outlets and three output outlets of the composed communication.

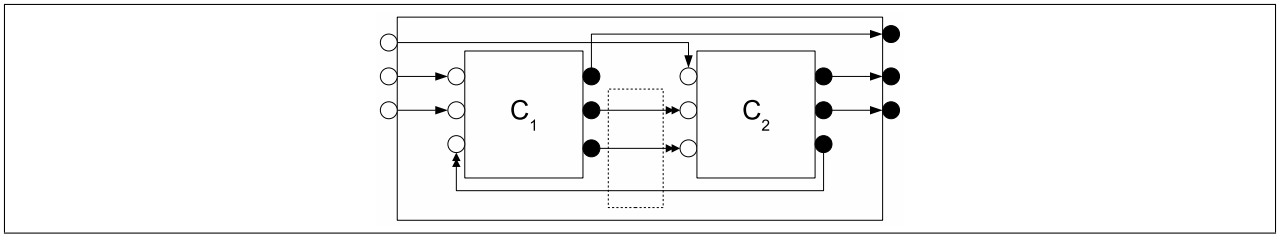

Figure 3.9: Example 2: Communication Composition

Example 3 Consider a contractor attempts to get the lowest price on a product from multiple vendors. Suppose the contractor divides those providers into two disjunct sets and sends two negotiators to negotiate

with vendors, one for each set. The reason for this arrangement may be the negotiation is too time-consuming to be completed by one negotiator, or location concerns, e.g., each negotiator is geographically close to their assigned set of vendors. Under this circumstance, the contractor creates two communications, each of which is a *negotiator*, and composes them by having them interact with each other. Consequently, the two negotiators can exchange the lowest price they obtained so far and then use this information to obtain a better offer from their assigned set of vendors. Figure 3.10 shows the relationship between these two negotiators. The visible outlets are used to interact with vendors.

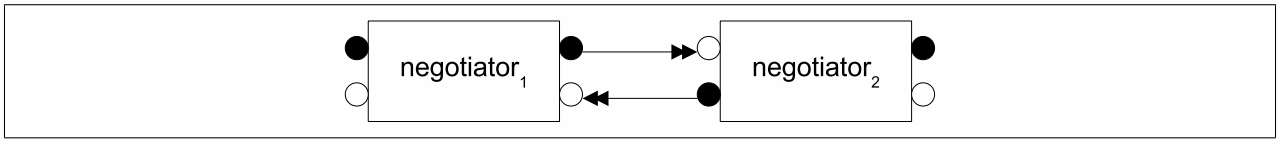

Figure 3.10: Example 3: Contract Negotiators

3.3.3 General Case

We have seen a number of composed communications. This section presents the general case of a composed communication shown in Figure 3.11. In the figure, we use vertical lines to represent compositions (using notation adopted from process calculi). C_1 and C_2 are composed first, then the composed communication is composed with another communication, and so on. An external observer can only see the input outlets and the output outlets of the composed communication.

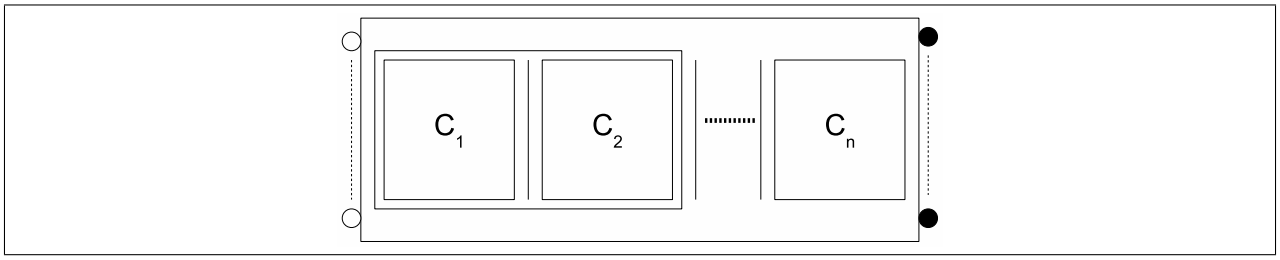

Figure 3.11: Composition: General Case

3.4 Summary

This chapter gives the definition for communications from a different angle, and presents three rules for constructing complex communications by composing simpler communications. A communication consists of a set of active objects, which are outlets and handlers. Through outlets, a communication interacts with computations by receiving messages from computations via its input outlets and by sending messages to computations via its output outlets. The received messages are handled by handlers, which sends the processed messages to output outlets.

A communication can be as simple as a *channel*, which transfers asynchronous messages from a process connecting to its input outlet to another process connecting to its output outlet. Because of the composition rules, we can build more complex communications starting from channels.

Not only a communication can be used as communication medium by a number of computations, but it can also be used by a single computation. At the sender side, a communication can be used as a *worker*, which applies rules to outgoing messages. Figure 3.12 shows this case. For instance, workers can delay, encrypt, decrypt, or type check messages sent by the sender.

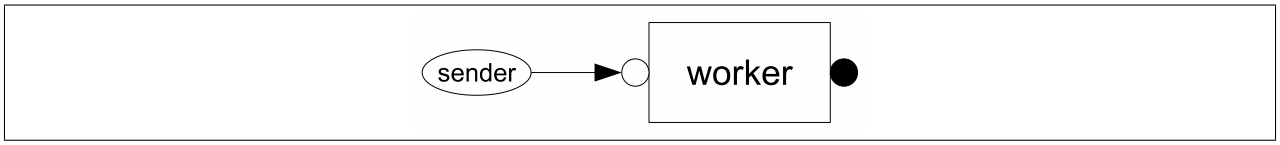

Figure 3.12: Communication as a Worker

At the recipient side, a communication can be used as a dedicated *concierge* applying rules to incoming messages. Figure 3.13 illustrates this case. For example, a concierge can be used as a filter and only certain types of messages will be delivered to the recipient.

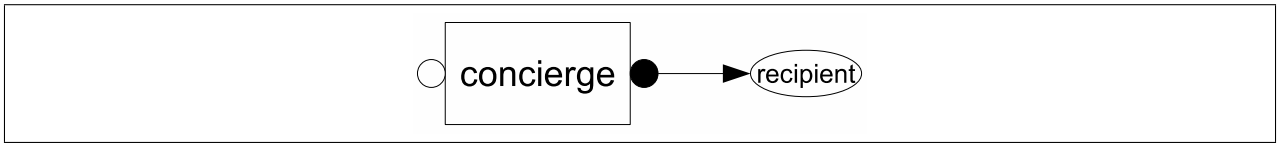

Figure 3.13: Communication as a Concierge

Using a communication as a *concierge* is not uncommon. For example, in Gmail, priority inbox [35] is used to prioritize incoming messages for email recipients. *Selectors* [41] is another example, in which a computation has multiple guarded mailboxes and the next message to be picked up for processing is dependent on the boolean guards on mailboxes. In the domain of Email systems, the concept of using a communication to enforce the recipients' receiving policies has been used for curbing spam [42].

Chapter 4

interActors

This chapter introduces *interActors* – a model for separating communication concerns of processes from their functional concerns. *interActors* are defined in terms of the actor model of concurrency and extend the actor model with support for complex communications between actors. I first introduce the basic concepts of *interActors* in Section 4.1. Section 4.2 introduces the actor model, the foundation underlying *interActors*. Section 4.3 presents operational semantics for *interActors*. Compositional semantics are presented in Section 4.4. Section 4.5 discusses the observational equivalence of communications and Section 4.6 concludes the chapter with discussions of some open issues.

4.1 Basic Concepts

interActors are defined in terms of the Actor model of concurrency [2, 4], and extend Actors with support for complex communications. In *interActors*, concurrent computations are represented by actors. A runtime system implementing *interActors* includes two layers: the Actor Layer and the Communication Layer (see Figure 4.1). In the figure, ovals represent actors; the rectangle is a communication; the white circle is the input outlet of the communication; the black circle is the output outlet of the communication. Communications reside in the Communication Layer and the actors carrying out the computations reside in the Actor Layer.

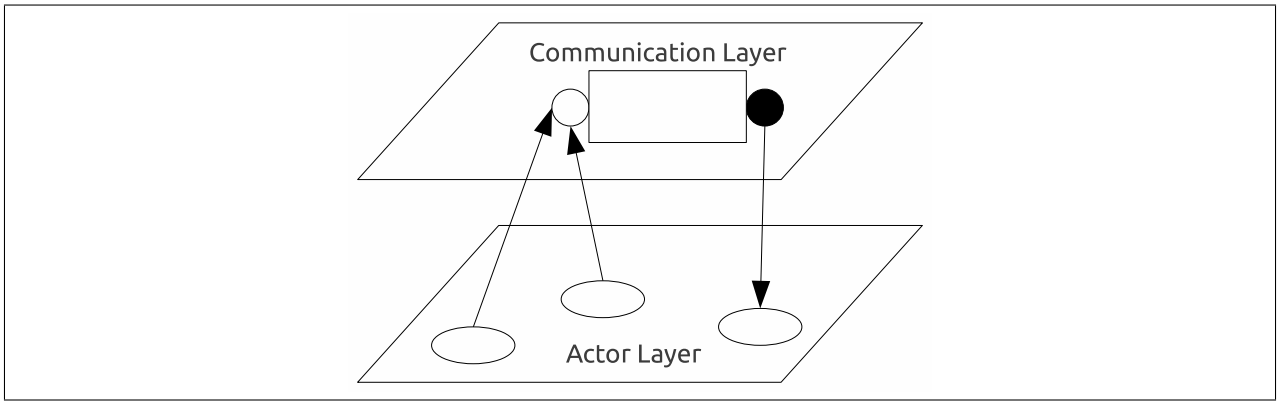

Figure 4.1: Two-layer Runtime System

An actor in the Actor Layer interacts with a communication only by sending messages to input outlets of the communication and receiving messages from the communication's output outlets. Handlers of a

communication are the driving force under the hood, and they can create more handlers, create more outlets, or change behaviors of outlets. Actors are oblivious to handlers and output outlets. In other words, actors only need to know the names of input outlets to participate in communications. Furthermore, because handlers can send messages to outlets, handlers need to know the names of outlets beforehand in order to send them messages.

4.2 Actors

Actors is a language-independent mathematical model for concurrent computing. Actors extend the concept of objects to concurrent computations. An actor (see Figure 4.2) is an active object encapsulating a thread of execution, a mailbox, a set of methods, and the object's states. An actor interacts with other actors through point-to-point asynchronous message passing.

Each actor has a globally unique name using which other actors communicate with it. In response to an incoming message, an actor may carry out one of three actor primitives:

- `send(a,v)` sends a message `v` to an actor named `a` asynchronously.
- `create(b, v)` creates a new actor with behavior `b` and initial parameter `v`, and returns a unique actor name of the newly created actor.
- `become(b)` creates an anonymous actor to complete the rest of the current computation, changes behavior of the actor executing `become(b)` to be `b` and frees that actor to accept new messages.

An actor system evolves as actors interact with each other by sending messages. The actor model guarantees weak-fairness. That is, messages are guaranteed to be eventually delivered to their destinations. Message delivery order, however, is not guaranteed. Messages delivered but not processed by the receiver are buffered in the mailbox of the receiver. Message processing is atomic, which means that once an actor starts processing a message, it continues non-preemptively until it is completed.

4.2.1 Syntax and Semantics

Instantaneous snapshots of actor systems are called *actor configurations*. Actor semantics are defined using a transition relation on configurations. The notion of open systems is captured by defining a dynamic interface for a configuration, i.e., by explicitly representing a set of *receptionists*, actors which may receive messages from actors outside the configuration and a set of *external* actors outside the configuration which may receive messages from actors inside the configuration.

An actor configuration with actor map α, a finite set of undelivered messages μ, receptionists ρ, and external actors χ, is written as

$$\langle \alpha \mid \mu \rangle_{\chi}^{\rho}$$

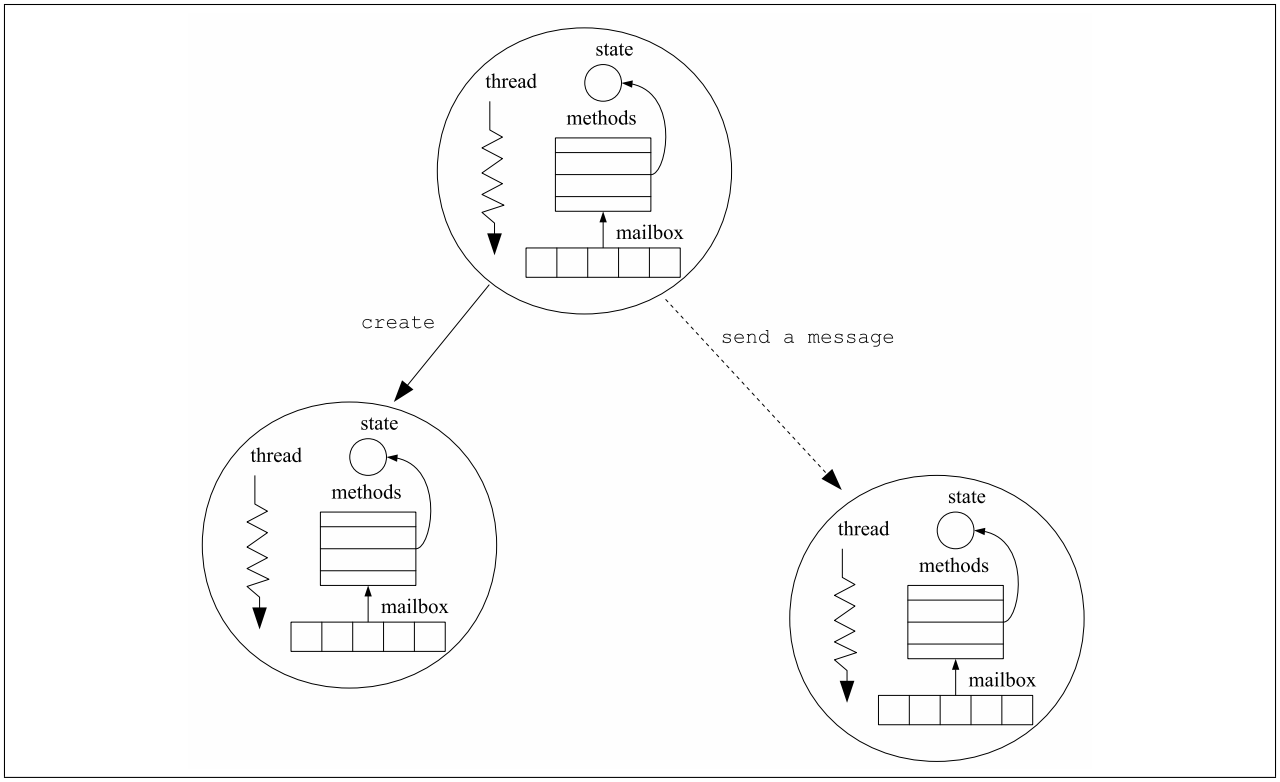

Figure 4.2: An actor may be viewd as an object augmented with a thread of control and a mailbox. Actors interact with each other through asynchronous messages and may create new actors.

where ρ and χ are finite sets of actor addresses, α maps a finite set of actor addresses to their behaviors, μ is a finite multi-set of undelivered messages. A message m which has two parts a target a and message content v, is written as $a \triangleleft v$. If $A = Dom(\alpha)$ (domain of α), then the following properties must hold:

(0) $\rho \subseteq A$ and $A \cap \chi = \phi$,

(1) if $a \in A$, then $FV(\alpha(a)) \subseteq A \cup \chi$, where $FV(\alpha(a))$ represents the free variables of $\alpha(a)$; and if $v_0 \triangleleft v_1$ is a message with content v_1 to actor address v_0, then $FV(v_i) \subseteq A \cup \chi$ for $i < 2$.

An actor a can be in one of two states: *idle* and *busy*.

- *idle* represented by $(b)_a$, which means that the actor is ready to accept a message, where b is its behavior
- *busy* represented by $[app(b, m)]_a$, which means that the actor is processing a message by applying its behavior b to the message m.

Transition Rules

An actor expression, e, is either a value v, or otherwise it can be uniquely decomposed into a reduction context, R, filled with a *redex*, r, denoted as $e = R[r]$. A *redex* represents the next sub-expression to evaluate in a standard left-first, call-by-value evaluation strategy.

In a reduction context R, the evaluation of the current expression occurs, and *app* is a function which, when evaluated, applies the received message to the behavior of the receiving actor. The transition rules on actor configurations are defined as follows.

The following rule says that if expression e reduces to expression e' in the context of $Dom(\alpha) \cup \{a\} \cup \chi$, then an actor a with behavior e will change its behavior to e':

$$e \xrightarrow{\lambda}_{Dom(\alpha)\cup\{a\}\cup\chi} e' \Rightarrow \langle \alpha, [e]_a \mid \mu \rangle^{\rho}_{\chi} \rightarrow \langle \alpha, [e']_a \mid \mu \rangle^{\rho}_{\chi}$$

The following rule defines the asynchronous semantics of message send, in which an actor a sends a message $m = a' \triangleleft msg$ to another actor named a':

$$\langle \alpha, [R[\![\texttt{send(m)}]\!]]_a \mid \mu \rangle^{\rho}_{\chi} \rightarrow \langle \alpha, [R[\![nil]\!]]_a \mid \mu' \rangle^{\rho}_{\chi}$$

where $\mu' = \mu \cup \{m\}$ and $m = a' \triangleleft msg$, a' is an actor and msg is the content of the sending message.

The following rule shows how an actor a can receive a message when it is idle. As a result, the actor applies the received message to its current behavior:

$$\langle \alpha, (b)_a \mid \mu, a \triangleleft msg \rangle^{\rho}_{\chi} \rightarrow \langle \alpha, [app(b, msg)]_a \mid \mu \rangle^{\rho}_{\chi}$$

The following rule shows how a new actor with *fresh* name a' is created and the new actor's name a' is returned to the creator a:

$$\langle \alpha, [R[\![\texttt{newactor(b)}]\!]]_a \mid \mu \rangle^{\rho}_{\chi} \rightarrow \langle \alpha, [R[\![a']\!]]_a, (b)_{a'} \mid \mu \rangle^{\rho}_{\chi}$$

where a' is the fresh name of the newly created actor.

Composition of Actor Configuration

Actor configurations can be composed to form new actor configurations. Two actor configurations $\langle \alpha_0 \mid \mu_0 \rangle^{\rho_0}_{\chi_0}$ and $\langle \alpha_1 \mid \mu_1 \rangle^{\rho_1}_{\chi_1}$ are *composable* if $Dom(\alpha_0) \cap Dom(\alpha_1) = \phi$, $\chi_0 \cap Dom(\alpha_1) \subseteq \rho_1$, and $\chi_1 \cap Dom(\alpha_0) \subseteq \rho_0$. The new composed configuration can be written as:

$$\langle \alpha_0 \cup \alpha_1 \mid \mu_0 \cup \mu_1 \rangle^{\rho_0 \cup \rho_1}_{(\chi_0 \cup \chi_1) - (\rho_0 \cup \rho_1)}$$

This composition operation is commutative, associative, and has the empty configuration as unit, which are discussed in [4].

4.3 interActors

In *interActors*, we use actors to model outlets and handlers. Furthermore, we need a binding to indicate which outlet is known to which handler so that handlers can send messages to outlets or change the behavior of outlets. Hence,

Definition (Communication):

A communication denoted as C can be written as $(H : O : P)$, where

- H is a set of handlers which are system actors and drive the communication forward.
- O is a set of outlets which are system actors including input outlets and output outlets.
- P is a set of mappings from handlers to outlets indicating which handler have names of which outlets. Each mapping has the form of (h, o), where h is the name of a handler and o is the name of an outlet.

Definition (interActor Configuration):
An instantaneous snapshot of a system of *interActors* is called an interActor configuration. An interActor configuration extends an actor configuration with a finite set of communications, a finite set of messages in the communication layer, and a finite set of mappings from application actors to input outlets. Because *interActors* do not alter how an actor configuration interacts with external actors, we do no show receptionists and external actors. Therefore, an *interActors* configuration can be represented by a 5-tuple:

$$\langle \mathcal{C} \mid \mathcal{M} \mid \mathcal{B} \mid \alpha \mid \mu \rangle$$

where $\mathcal{C}$ is a finite set of communications. $\mathcal{M}$ is a finite set of messages in the communication layer. $\mathcal{B}$ is a set of mappings from application actors to input outlets, and each mapping indicates which application actor has the name of which input outlet. Each mapping has the form of (a, o) and a is the name of an application actor and o is the name of an outlet. α and μ are similar to identical symbols in an actor configuration. α maps actor addresses to behaviors, and μ is a finite set of undelivered messages to and from application actors.

Next, I present transition rules of *interActors*.

Launching a communication

Once an application actor has a communication, `comm`, which has been created by itself or received from another actor, it can launch it by calling `launch(comm)`. As a consequence, in the communication layer, a communication is created along with its outlets and handlers. Application actors are given the names of the input outlets of the communication. The following transition describes this operation.

$$\langle \mathcal{C} \mid \mathcal{M} \mid \mathcal{B} \mid \alpha,\ [R[\![\texttt{launch(comm)}]\!]_a \mid \mu \rangle \rightarrow \langle \mathcal{C}, (H : O : P) \mid \mathcal{M} \mid \mathcal{B} \mid \alpha,\ [R[\![nil]\!]_a \mid \mu' \rangle$$

where a is the application actor launching the communication, and $\mu' = \mu \cup m$, where m is a set of messages notifying application actors the names of the input outlets.

Figure 4.3 illustrates this transition. In the figure, an actor represented by the oval calls `launch` to launch a communication, which leads to a new communication is created in the communication layer.

Actor sending a message to an outlet

An actor a sends a message to an outlet o using `send`. More specifically, an actor sends a message to an input outlet, because, actors only know input outlets but are oblivious to output outlets. The result of this

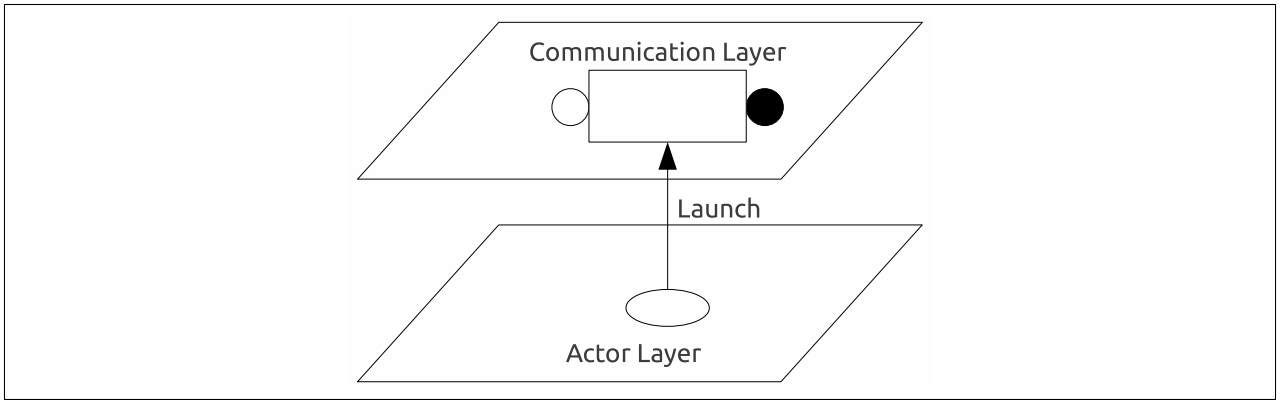

Figure 4.3: Launch a Communication

operation is an actor message targeting to o is created in the actor layer. The transition is depicted as follow:

$$\langle \mathcal{C} \mid \mathcal{M} \mid \mathcal{B} \mid \alpha, [R[\![\texttt{send(m)}]\!]_a \mid \mu \rangle \rightarrow \langle \mathcal{C} \mid \mathcal{M} \mid \mathcal{B} \mid \alpha, [R[\![nil]\!]_a \mid \mu' \rangle$$

where $\mu' = \mu \cup \{m\}$, and $m = o \triangleleft msg$ and o is an input outlet.

Actor receiving outlet name

To participate in a communication, an actor has to send messages to the communication's input outlet. Actor a receives a message which has the name of an outlet o so that it can use the received outlet to interact with the communication. Again, the received outlet is the type of *input*, because actors do not use output outlets to send messages. This transition is written as

$$\langle \mathcal{C} \mid \mathcal{M} \mid \mathcal{B} \mid \alpha, (b)_a \mid \mu, a \triangleleft o \rangle \rightarrow \langle \mathcal{C} \mid \mathcal{M} \mid \mathcal{B}' \mid \alpha, [app(b, o)]_a \mid \mu \rangle$$

where o is an outlet, and $\mathcal{B}' = \mathcal{B} \cup \{(a, o)\}$. *app* is a function which applies the received message to the behavior of the actor. As a result of this operation, a new mapping from actor a to outlet o is added into $\mathcal{B}$, and a's behavior b is applied to the incoming outlet name o to record it.

Figure 4.4 illustrates this case. In the figure, the line with the arrow from a to o represents the mapping from a to o. Note that the received outlet name can be from either another actor or a communication.

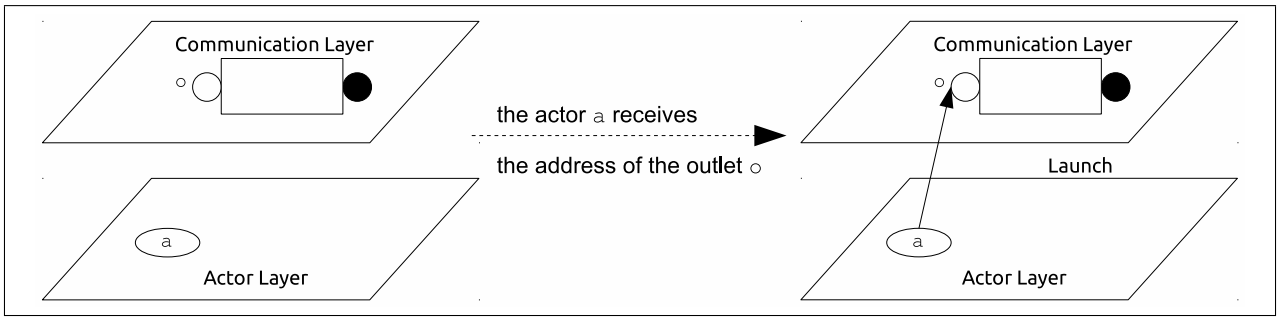

Figure 4.4: An Actor Receives an Outlet

Outlet sending message

An input outlet can sends a message to a handler or an output outlet, and an output outlet can send a message to an application actor. In the former case, a message is created in the communication layer; in the latter case, a message is generated in the actor layer. This scenario is depicted in the following:

$$\langle \mathcal{C},\ (H : O, [R[\![\texttt{send(m)}]\!]_o : P) \mid \mathcal{M} \mid \mathcal{B} \mid \alpha \mid \mu \rangle \rightarrow \langle \mathcal{C},\ (H : O, [R[\![nil]\!]_o : P) \mid \mathcal{M}' \mid \mathcal{B} \mid \alpha \mid \mu' \rangle$$

where o is the outlet. $m = target \triangleleft msg$ and $target$ is a handler, an output outlet, or an application actor. $\mathcal{M}' = \mathcal{M} \cup \{m\}$ and $\mu' = \mu$ if o is an input outlet, and $\mathcal{M}' = \mathcal{M}$ and $\mu' = \mu \cup \{m\}$ if o is an output outlet.

Outlet receiving message

An input outlet can receive messages from application actors, and both input outlets and output outlets can receive messages from handlers. The receiving outlet applies the received message to its current behavior, which is depicted as follow:

$$\langle \mathcal{C},\ (H : O, (b)_o : P) \mid \mathcal{M} \mid \mathcal{B} \mid \alpha \mid \mu \rangle \rightarrow \langle \mathcal{C},\ (H : O, [app(b, msg)]_o : P) \mid \mathcal{M}' \mid \mathcal{B} \mid \alpha \mid \mu' \rangle$$

where o is the outlet with behavior b. The message is removed from $\mathcal{M}$ or μ depending on whether it was from a handler or an application actor, respectively. $\mathcal{M}' = \mathcal{M}$ and $\mu' = \mu - \{o \triangleleft msg\}$ if the sender is an actor, and $\mathcal{M}' = \mathcal{M} - \{o \triangleleft msg\}$ and $\mu' = \mu$ if the sender is a handler. *app* applies o's behavior b to the message.

Changing outlet's behavior As a special case of this rule, if msg is the type of $behv(b')$, which means a handler is trying to change the behavior of the outlet to b', applying b to it changes the behavior of o to b': $[app(b, msg)]_o = (b')_o$. These special messages can only be sent by handlers, and application actors have no such privilege.

Handler sending message

A handler sends a message to another handler or to an outlet using `send`. As a result, a new message is created in the communication layer.

$$\langle \mathcal{C},\ (H, [R[\![\texttt{send(m)}]\!]_h : O : P) \mid \mathcal{M} \mid \mathcal{B} \mid \alpha \mid \mu \rangle \rightarrow \langle \mathcal{C},\ (H : [R[\![nil]\!]_h : O : P) \mid \mathcal{M}' \mid \mathcal{B} \mid \alpha \mid \mu \rangle$$

where h is the handler sending a message, $\mathcal{M}' = \mathcal{M} \cup \{m\}$, and $m = target \triangleleft msg$ and $target$ is a handler or an outlet in the same communication.

Handler receiving message

A handler can receive a message from an outlet or another handler. As a consequence of receiving a message, the handler applies it to its current behavior. Furthermore, if the message contains a name of an outlet not

known to the handler, a new mapping from the handler to the outlet is added to P. The following describes this case.

$$\langle \mathcal{C},\ (H, (b)_h : O : P) \mid \mathcal{M}, h \triangleleft msg \mid \mathcal{B} \mid \alpha \mid \mu \rangle \rightarrow \langle \mathcal{C},\ (H, [app(b, msg)]_h : O : P') \mid \mathcal{M} \mid \mathcal{B} \mid \alpha \mid \mu \rangle$$

where h is a handler with behavior b. $P' = P \cup \{(h, outlet\_name(msg))\}$ if msg contains a name of an outlet which is unknown to the handler, where *outlet_name* is a function which retrieves an outlet name from msg; $P' = P$ otherwise.

Handler creating outlet

A handler can create a new outlet through invoking `new_outlet`. At creation, only the creator knows the name of the created outlet. Subsequently, the creator can send the newly created outlet's name to other handlers, or application actors if the created outlet is the type of *input*. This operation is shown in the following:

$$\langle \mathcal{C},\ (H, [R[\![\texttt{new\_outlet}\ (b)\ ]\!]]_h : O : P)_c \mid \mathcal{M} \mid \mathcal{B} \mid \alpha \mid \mu \rangle$$
$$\rightarrow \langle \mathcal{C},\ (H, [R[\![o]\!]]_h : O, (b)_o : P')_c \mid \mathcal{M} \mid \mathcal{B} \mid \alpha \mid \mu \rangle$$

where h is the creating handler, b is the behavior of the newly created outlet, o is fresh and is the name for the newly created outlet, and $P' = P \cup \{(h, o)\}$ indicating h's knowledge of o.

Figure 4.5 illustrates this case. The line from h to o indicates h's knowledge of o.

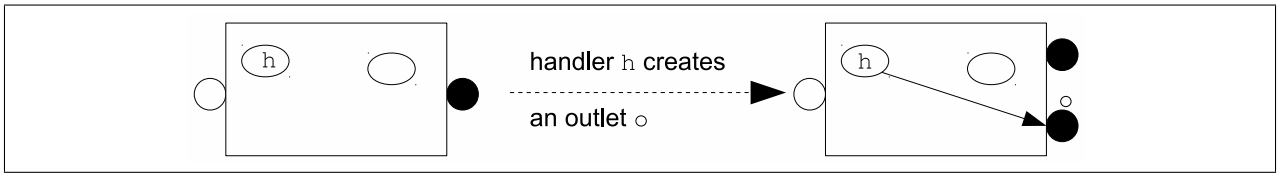

Figure 4.5: An Handler Creates an Outlet

Handler creating another handler

A handler can create a new handler through invoking `new_handler`. The newly created handler is only known to the creator, which may share the name to other handlers and outlets. This operation is described below:

$$\langle \mathcal{C},\ (H, [R[\![\texttt{new\_handler}\ (b)\ ]\!]]_h : O : P)_c \mid \mathcal{M} \mid \mathcal{B} \mid \alpha \mid \mu \rangle$$
$$\rightarrow \langle \mathcal{C},\ (H, [R[\![h']\!]]_h, (b)_{h'} : O : P)_c \mid \mathcal{M} \mid \mathcal{B} \mid \alpha \mid \mu \rangle$$

where h is the creating handler, b is the behavior of the newly created handler, and h' is fresh and is the name for the new handler. h' is known to h – the creator.

Figure 4.6 illustrates this case. The line from h to h' indicates h's knowledge of the newly created handler h'.

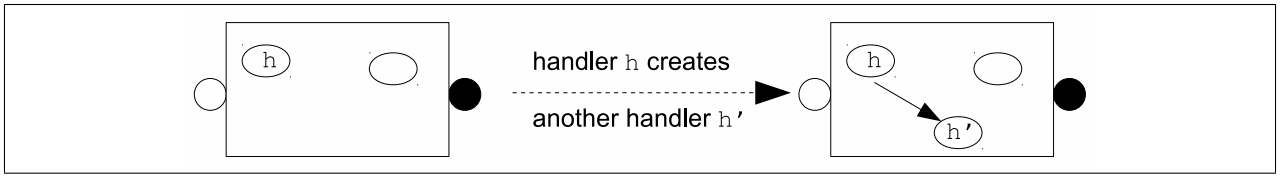

Figure 4.6: An Handler Creates another Handler

4.4 Compositional Semantics

This section presents the communication compositional semantics. We denote a communication C as $(H : O : P)$ where H is a set of handlers, O is a set of outlets, and P is a set of mappings from handlers to outlets, each of which has the form of (h, o). A handler with name h, behavior b, target t is represented by $(b, t)_h$; an outlet with name o, behavior b, target t is represented by $(b, t)_o$.

Input Merge

To merge a number of communications at their input ends, we receive as parameters a set of input outlets to be merged and a behavior. The input outlets are transformed into handlers which retain the old behaviors and targets of the outlets. Then we create a new handler with the provided behavior and a new input outlet; the input outlet becomes the input outlet of the composed communication, and forwards whatever it receives to the handler, and the handler's targets are the new handlers transformed from the composed input outlets.

The input merge of n $(n > 1)$ communications can be described as follow:

$$\underset{S,b}{|}{}_{i=1}^{n}(H_i : O_i : P_i) \Rightarrow \Big(\bigcup_{i=1}^{n} H_i \cup (b, T)_h \cup T : (\bigcup_{i=1}^{n} O_i - S) \cup \{(forwarder, h)_{o'}\} : (\bigcup_{i=1}^{n} P_i - P_M) \cup P_N \} \Big)$$

where $n > 1$, S is a set of input outlets to be composed, b is the provided behavior, $(b, T)_h$ is the newly created handler with behavior b and targeting T, where $T = \bigcup_{o \in S} \{tr((b_o, t_o)_o)\}$ is a set of handlers transformed from outlets in S using a transform function tr, which retains the original outlet's behavior and its targets; $(forwarder, h)_{o'}$ is the newly created input outlet having behavior *forwarder* and targeting h, the newly created handler; The map tracking handlers' knowledge of outlets is obtained by including the mappings of the composing communications $\bigcup_{i=1}^{n} P_i$, removing the mappings involving transformed outlets P_M, and adding the newly created mapping P_N.

Figure 4.7 illustrates this with an example where three communications, C_1, C_2, and C_3, are to be composed using the set $\{in_1, in_2, in_3\}$ and behavior b. A new input outlet is created which targets the newly created handler represented as the gray oval, which has the behavior b. The composed communication has one input outlet and three output outlets, which are the only outlets visible to external observers. The merged input outlets are transformed into handlers (the three white ovals) which have the same behaviors and the same targets as the original input outlets.

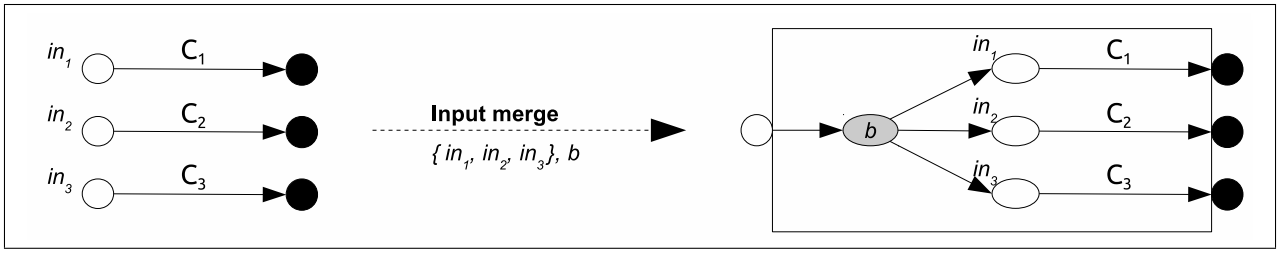

Figure 4.7: Input Merge Composition

Note that *input merge* also allows the merged input outlets from a same communication. Figure 4.8 shows an example. In this figure, two communication C_1 and C_2 are composed. The set of merged input outlets is $\{in_1, in_2, in_3\}$. Among them, the first two outlets are from C_1 and the third outlet is from C_2. The supplied behavior is b. in_5 is the input outlet created for this composition.

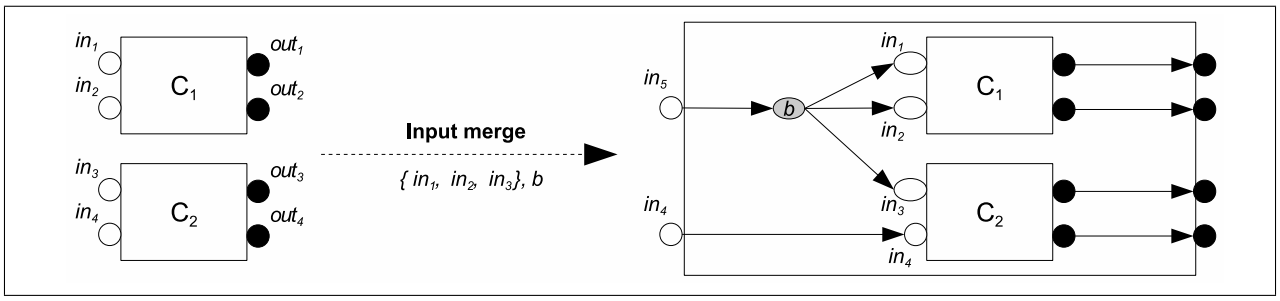

Figure 4.8: Input Merge Composition – Two Outlets from Same Communication

Output Merge

To compose a number of communications by merging their output outlets, we receive as parameters the set of output outlets to be merged, one behavior, and one target. We create a new output outlet with *forwarder* behavior targeting the provided target, create a handler with the provided behavior targeting the newly created output outlet, and finally transform the output outlets to be merged into handlers which have the newly created handler as their target.

This output merge of n ($n > 1$) communications can be described as follow:

$$\underset{S,b,t}{|}\ {}^{n}_{i=1}(H_i : O_i : P_i) \Rightarrow \Big(\bigcup_{i=1}^{n} H_i \cup (b, o')_h \cup T : (\bigcup_{i=1}^{n} O_i - S) \cup \{(forwarder, t)_{o'}\} : (\bigcup_{i=1}^{n} P_i - P_M) \cup P_N \Big)$$

where $n > 1$, S is a set of output outlets to be composed, b is the provided behavior, $(forwarder, t)_{o'}$ is the newly created output outlet with behavior *forwarder* targeting the provided target t, and $(b, o')_h$ represents a handler named h with the behavior b targeting o'; $T = \bigcup_{o \in S} \{tr((b_o, h)_o)\}$ is a set of handlers transformed from S using a transform function tr and each handler retains the same behavior and they all have their target to the newly created handler h. The map tracking handlers' knowledge of outlets is obtained by including

the mappings of the composing communications $\bigcup_{i=1}^{n} P_i$, removing the mappings involving transformed outlets P_M, and adding the newly created mapping P_N.

This is illustrated in Figure 4.9, in which C_1, C_2, and C_3 are merged under the set $\{out_1, out_2, out_3\}$, behavior b, and targets t. Note that t is a list of recipients. The gray oval represents the newly created handler having behavior b, which sends its processed messages to the newly created output outlet, which targets t. The three black ovals are handlers transformed from the merged output outlets. The composed communication has three input outlets and one output outlet, which are the only outlets visible to external observers.

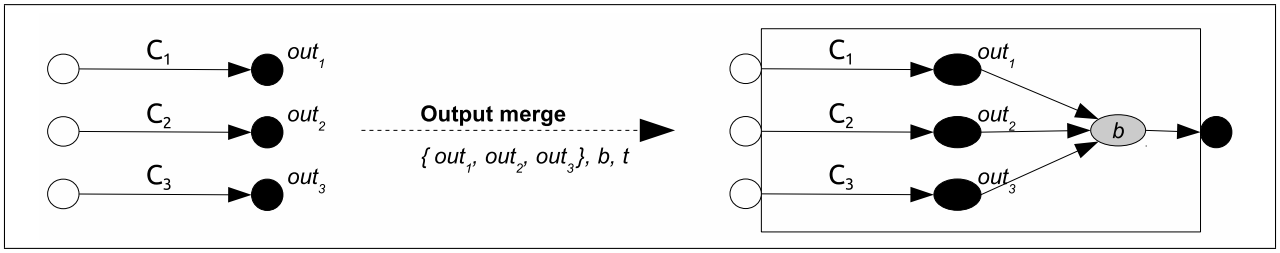

Figure 4.9: Output Merge Composition

Like *input merge*, *output merge* does not rule out the case that the merged output outlets can be from a same communication. Figure 4.10 shows an example. In this figure, the set of merged output outlets are out_2, out_3, and out_4, and the first outlet is from C_1 and the last two outlets are from C_2. The supplied behavior is b and the newly created output outlet out_5 has targets to t.

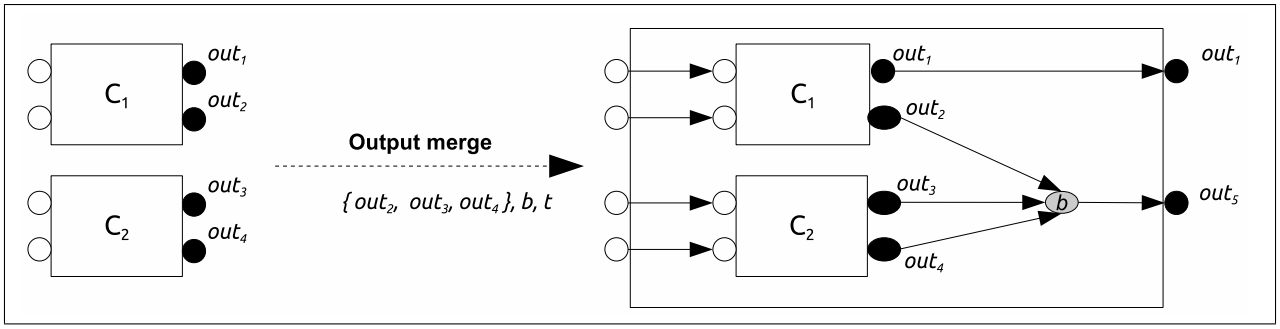

Figure 4.10: Output Merge Composition – Outlets from a Same Communication

Output-Input Merge

Communications can be composed using *output-input merge*, which connects output outlets of one communication to input outlets of another communication. Therefore, the message flow is from the first communication to the second communication. To compose a number of communications by connecting output outlets to input outlets, bindings must be provided, each of which has the form (out, in, b), where out is the name of an output outlet, in is the name of an input outlet, and b is the behavior for the newly created handler which

is the bridge between *out* and *in*. The *output-input merge* composition of n ($n > 1$) communications can be described as follow.

$$\underset{B}{|}_{i=1}^{n}(H_i : O_i : P_i) \Rightarrow \Big(\big(\bigcup_{i=1}^{n} H_i \big) \cup \bigcup_{j=1}^{m} \{(b_j, in_j)_{h_j}\} \cup \bigcup_{j=1}^{m} \{tr((bo_j, h_j)_{out_j})\} \cup \bigcup_{j=1}^{m} \{tr((bi_j, ti_j)_{in_j})\} : \big(\bigcup_{i=1}^{n} O_i \big) - \big(\bigcup_{j=1}^{m} \{(bo_j, to_j)_{out_j}\} \big) - \big(\bigcup_{j=1}^{m} \{(bi_j, ti_j)_{in_j}\} \big) : \big(\bigcup_{i=1}^{n} P_i - P_M \big) \Big)$$

where $n > 1$; B is a provided set of bindings, with its j^{th} entry has the form of $((bo_j, to_j)_{out_j}, (bi_j, ti_j)_{in_j}, b_j)$, which is a triple identifying an output outlet out_j, which has the behavior of bo_j and the target of to_j, an input outlet in_j, which has the behavior of bi_j and the target of ti_j, and a behavior b_j, where $1 \leq j \leq m$ and m is the number of entries in B. To compose the communications, for each binding triple (out_j, in_j, b_j) in B, both outlets are transformed into handlers, and a new handler is created to connect them. $tr((bo_j, h_j)_{out_j})$ is the handler transformed from the output outlet $(bo_j, to_j)_{out_j}$ which has the behavior of bo_j but the target is changed from to_j to the newly created handler h_j; $tr((bi_j, ti_j)_{in_j})$ is the handler transformed from the input outlet $(bi_j, ti_j)_{in_j}$ which has the behavior of bi_j and the original target of ti_j. $(b_j, in_j)_{h_j}$ is the handler, which connects the two transformed handlers, with the provided behavior b_j targeting in_j. The map tracking handlers' knowledge about outlets is obtained by including the mappings of the composed communications $\bigcup_{i=1}^{n} P_i$ and removing the mappings involving transformed outlets P_M. No new handler to outlet mappings are created in this composition.

Figure 4.11 illustrates this using the example of two communications, C_1 and C_2, to be composed using the binding set $\{(out_2, in_3, b_1), (out_4, in_2, b_2)\}$. C_1 has two input outlets in_1 and in_2 and two output outlets out_1 and out_2, and C_2 has two input outlets in_3 and in_4 and two output outlets out_3 and out_4. The *output-input merge* composition creates two handlers represented by the two gray ovals, which have behaviors b_1 and b_2, respectively. The white ovals are handlers transformed from the merged input outlets and the black ovals are handlers transformed from the merged output outlets. To external observers, only the input outlet in_1 and the output outlet out_1 from C_1 and the input outlet in_4 and the output outlet out_3 from C_2 are visible, and those merged outlets are hidden from them.

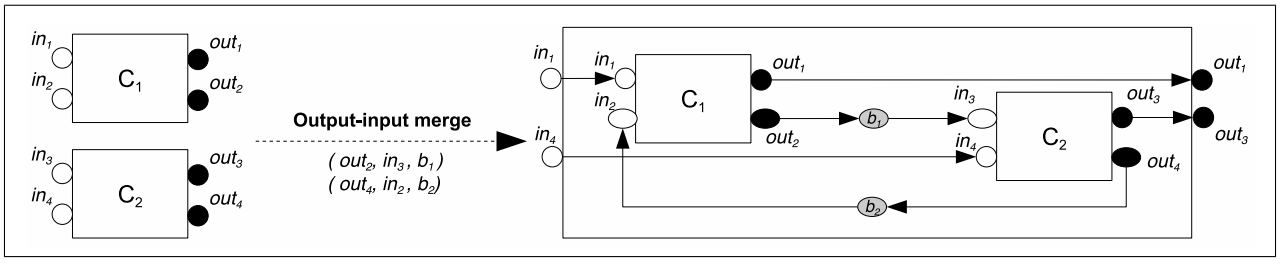

Figure 4.11: Output Input Merge Composition

output-input merge does not prevent multiple different output outlets are merged with a same input outlet. Figure 4.12 shows an example. This figure has one more binding (out_1, in_3, b_3) which merges the

output outlet out_1 of C_1 with the input outlet in_3 of C_2. In this case, both output outlets of C_1 send messages to in_3. Because the behavior of in_3 does not change, the two streams from out_1 and out_2 are simply merged into one stream at in_3.

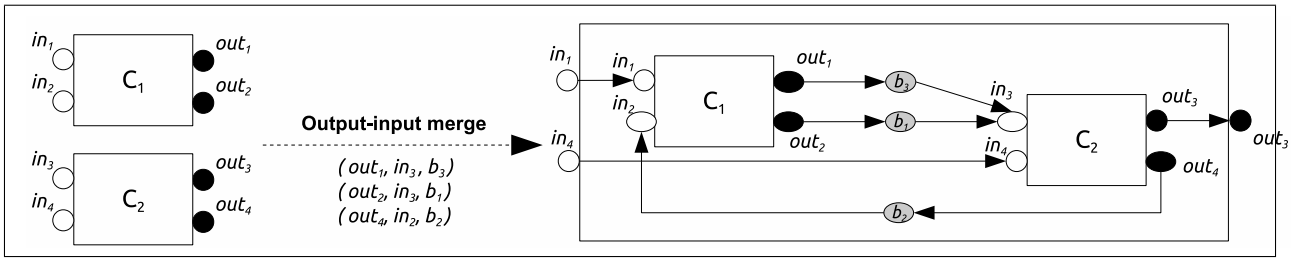

Figure 4.12: Output-input Merge Composition – Multiple Output Outlets to One Input Outlet

Likewise, *output-input* merge does not prevent one output outlet is merged with multiple input outlets. Figure 4.13 illustrates this case. In this figure, binding (out_2, in_4, b_3) is added. In this case, messages received at out_2 will be sent to in_3 and in_4 at the same time.

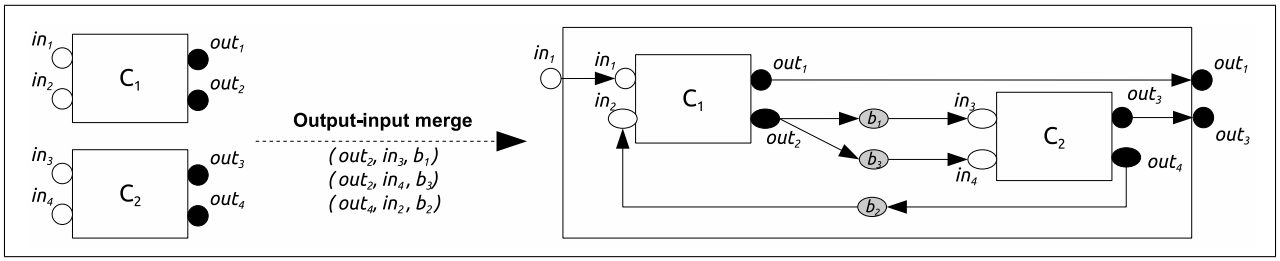

Figure 4.13: Output-input Merge Composition – One Output Outlet to Multiple Input Outlets

4.5 Equivalence of Communication Configurations

Now we introduce the notion of observational equivalence of communication configurations. Because a communication contains handlers and outlets – which are actors – and messages transferred to outlets and handlers, it is essentially an actor system, in which the input outlets known to application actors are receptionists, and the application actors known to the output outlets are external actors.

Definition (Communication Configuration):
A communication configuration can be written as:

$$\langle \Gamma \mid \Delta \rangle_o^i$$

where Γ maps outlets' and handlers' addresses to behaviors, Δ is a finite set of undelivered messages to the communication's outlets and handlers, i is a set of input outlets known to applications actors, and o is a set of application actors known to the output outlets of the communication.

Agha et al. discussed the equivalence of actor configurations in [4]. Because communication configurations are also actor configurations, we can use their approach to show the equivalence of communication

configurations. In this thesis, I use ideas similar to those presented in their paper, and propose an approach specifically for communication configurations.

To check whether two communication configurations are observationally equivalent, we need to put them into an "observing context." An observing context is some complete program with a hole, such that all of the free variables in an expression being observed are captured when the expression is placed in the hole. We define observing configurations for a communication configuration as configurations of the system of application actors.

Definition (Observing Configurations):
The observing configurations for a communication configuration, $\Omega = \langle \Gamma \mid \Delta \rangle_o^i$, are actor configurations of the form $\kappa = \langle \alpha \mid \mu \rangle_i^o$. For these actor configurations, receptionists are o and external actors are i. κ is composable with Ω.

Two communication configurations are considered to be equivalent if they behave the same way when they are placed in any observing configuration. Specifically, to show that two communications Ω_1 and Ω_2 are observationally equivalent, we use an arbitrary actor configuration κ as an observing configuration. From κ, an idealized observer observes Ω_1 and Ω_2. If, from any observing configuration, the observer cannot distinguish Ω_1 from Ω_2, we say that the two communications are observationally equivalent. To achieve this goal, we introduce an observer primitive for observing configurations, `event`, and observe whether `event` occurs in any given expression.

Informally, given any arbitrary observing configuration, an idealized observer separately observes two communication configurations. If for both communication configurations, the same event is always observed in the given observing configuration, we say that the two communication configurations are observationally equivalent. Formally,

Definition ($\Omega_0 \equiv \Omega_1$):
if $\Omega_0 = \langle \Gamma_0 \mid \Delta_0 \rangle_{o_0}^{i_0} \equiv \Omega_1 = \langle \Gamma_1 \mid \Delta_1 \rangle_{o_1}^{i_1}$, the following three properties must hold

(1) $|i_0| = |i_1|$, i.e., they have the same number of receptionists, i.e., the input outlets known to application actors,

(2) there is a one-to-one mapping between i_0 and i_1; i.e., for each input outlet in i_0, there is exactly one corresponding input outlet in i_1, and vice versa,

(3) for any expression in any observing configuration, if we replace the names of input outlets in i_0 with the corresponding names of input outlets in i_1, the same event will be observed, and vice versa.

4.5.1 Properties

This section studies properties of equivalence. In the following, we use Ω_0, Ω_1, and Ω_2 to represent $\langle \Gamma_0 \mid \Delta_0 \rangle_{o_0}^{i_0}$, $\langle \Gamma_1 \mid \Delta_1 \rangle_{o_1}^{i_1}$, and $\langle \Gamma_2 \mid \Delta_2 \rangle_{o_2}^{i_2}$, respectively.

Property (Transitive):

If $\Omega_0 \equiv \Omega_1$ and $\Omega_1 \equiv \Omega_2$, then $\Omega_0 \equiv \Omega_2$.

Proof. By definition of equivalence. □

Property (Commutative):

$\Omega_0 \mid \Omega_1 \equiv \Omega_1 \mid \Omega_0$

Proof. By definition of the composition operator. □

Property (Associative):

$\Omega_0 \mid \Omega_1 \mid \Omega_2 \equiv (\Omega_0 \mid \Omega_1) \mid \Omega_2 \equiv \Omega_0 \mid (\Omega_1 \mid \Omega_2)$

Proof. First, by definition of the composition operator, which is left-associative, $\Omega_0 \mid \Omega_1 \mid \Omega_2 \equiv (\Omega_0 \mid \Omega_1) \mid \Omega_2$. Second, we prove $\Omega_0 \mid \Omega_1 \mid \Omega_2 \equiv \Omega_0 \mid (\Omega_1 \mid \Omega_2)$ as follows using commutative property:

$$\begin{aligned} \Omega_0 \mid \Omega_1 \mid \Omega_2 &\equiv \Omega_2 \mid \Omega_0 \mid \Omega_1 \\ &\equiv \Omega_1 \mid \Omega_2 \mid \Omega_0 \\ &\equiv (\Omega_1 \mid \Omega_2) \mid \Omega_0 \\ &\equiv \Omega_0 \mid (\Omega_1 \mid \Omega_2) \end{aligned}$$

Third, by transitive property, because $\Omega_0 \mid \Omega_1 \mid \Omega_2 \equiv (\Omega_0 \mid \Omega_1) \mid \Omega_2$ and $\Omega_0 \mid \Omega_1 \mid \Omega_2 \equiv \Omega_0 \mid (\Omega_1 \mid \Omega_2)$, $(\Omega_0 \mid \Omega_1) \mid \Omega_2 \equiv \Omega_0 \mid (\Omega_1 \mid \Omega_2)$

□

Definition (Composition Operator $\mid$):

Composition operator $\mid$ represents one of the composition rules defined in Section 4.4 and it is left-associative and commutative. $\Omega_0 \mid \Omega_1$ means that the configurations Ω_0 and Ω_1 are composed using input merge, output merge, or input-output merge.

Now, we look at a number of properties with respect to the equivalence of communication composition.

Property (Composition Equivalence):

If $\Omega_0 \equiv \Omega_1$, then $\Omega_0 \mid \Omega_2 \equiv \Omega_1 \mid \Omega_2$.

Proof. We need to show that this property holds for all three composition rules defined in Section 4.4. Therefore, there are three cases. We assume that for each composition $\Omega_0 \mid \Omega_2$ and $\Omega_1 \mid \Omega_2$, they use the same parameters for each composition rule.

- Case 1 (input merge): Suppose for both $\Omega_0 \mid \Omega_2$ and $\Omega_1 \mid \Omega_2$, the set of input outlets of Ω_2 to be merged is s_2; for $\Omega_0 \mid \Omega_2$, the set of input outlets of Ω_0 to be merged is s_0; for $\Omega_1 \mid \Omega_2$, the set of input outlets of Ω_1 to be merged is s_1; there is one-to-one mapping between s_0 and s_1. For $\Omega_0 \mid \Omega_2$, the newly created input outlet is o', then, according to the input-merge rule, the set of input outlets of the composed configuration is $S_0 = ((i_0 \cup i_2) - (s_0 \cup s_2)) \cup \{o'\}$. Similarly, for $\Omega_1 \mid \Omega_2$, if the newly created input outlet

is o'', then, the set of input outlets of the composed configuration is $S_1 = ((i_0 \cup \imath_2) - (s_1 \cup s_2)) \cup \{o''\}$. Because there is a one-to-one mapping between s_0 and s_1, $|s_0 \cup s_2| = |s_1 \cup s_2|$, therefore, $|S_0| = |S_1|$, satisfying condition 1 in the equivalence definition. By the same reason, we can easily establish one-to-one mapping between S_0 and S_1, meeting condition 2. To show condition 3 is also satisfied, suppose the newly created handler is h_0 for the first composition and h_1 for the second composition. Because h_0 and h_1 connect to $s_0 \cup s_1$ and $s_0 \cup s_2$, respectively and they have the same behavior, if one event is observed in $\Omega_0 \mid \Omega_2$, the same event will be observed in $\Omega_1 \mid \Omega_2$. Therefore, $\Omega_0 \mid \Omega_2 \equiv \Omega_1 \mid \Omega_2$.

- Case 2 (output merge): Similar analysis applies to this case.
- Case 3 (input-output merge): Similar analysis applies to this case for condition 1 and condition 2. For condition 3, instead of creating one handler, multiple handlers may be created, each for each binding. Because for each newly created handler in the first composition, the corresponding newly created handler in the second composition has the same behavior, if one event is observed in the first composed configuration, the same event will be observed in the second composed configuration. Therefore, $\Omega_0 \mid \Omega_2 \equiv \Omega_1 \mid \Omega_2$.

□

4.6 Discussion

Openness We can apply the notion of open systems in the actor model to the *interActors* model. Openness in the *interActors* model is supported as follows:

- New input outlets can be added dynamically by creating new input outlets.
- New actor names can be known to output outlets by sending those actor names to them by input outlets or handlers.
- The targets of outlets can be changed.
- The behaviors of outlets can be changed at run-time by handlers.

Garbage Collection Once launched, like actors, a communication is either busy or idle. Some idle communications may not be in use anymore. To prevent the run-time system from being filled with these unused communications, garbage collection can be triggered under certain circumstances. For example, a communication can be garbage collected if it terminates. To determine whether a communication is terminated, the following two conditions must be satisfied:

1. All handlers are idle and there are no messages in the queues of outlets and handlers of the communication.
2. No application actors have the name of the input outlets of the communication.

Communication Types Because it is simply data flowing between composed communications, *interActors* do not carry out type-checking when composing communications. If type-checking were required, it could be realized in the behaviors of outlets or handlers.

Privacy and Security The main focus of *interActors* is separation of communication concerns of applications from their functional concerns. Privacy and security concerns are not directly addressed.

Although this thesis does not address privacy concerns, the approach may be used to enhance privacy and security. For example, public-key encryption can be encapsulated in a communication. Computations' public keys can be passed to the communication at the time of instantiating the communication. The implementation of encryption and decryption behaviors required for such a communication is described in Section 5.2.2.

Chapter 5

Implementation

I have implemented a proof-of-concept prototype which can be used to program new types of communications. Our implementation is in Scala using the Akka actor library [75]. As I mentioned in earlier chapters, a communication consists of outlets and handlers which are active objects. In our implementation, we treat a communication as an object that holds references to outlets and handlers which are actors. Furthermore, outlets and handlers have behaviors.

The rest of this chapter is organized as follows: Section 5.1 presents the architecture of an *interActors* system. Section 5.2 describes the classes implemented for *interActors*: `CSL`, `Behavior`, `Handler`, `Outlet`, and `Communication`. Section 5.3 gives a number of communication examples. Section 5.4 introduces communication-oriented programming, that is, how to use communications in an application. The implementation of communication composition is discussed in Section 5.5. Finally, Section 5.6 summarizes this chapter and reveals a possible optimization for this prototype.

5.1 System Architecture

An *interActors* system is described in Figure 5.1. The system include three layers: application layer, communication layer, and system layer. The application layer is where the application actors reside, the communication layer deals with complex communications, and the system layer provides system-level services. In the figure, only message transport service is presented in the system layer because our focus is communication. Application actors can interact with each other through either message transport service offered by the system or through the communication layer. More precisely, for simple point-to-point messages, application actors use system-provided message services, and for more complex communication, application actors have a choice to use communications – a higher-level abstraction.

5.2 Classes

A runtime *interActors* system is made up of a number of classes: `CSL`, `Behavior`, `Handler`, `Outlet`, and `Communication`. The relationship between these classes is illustrated in Figure 5.2. `CSL` implements methods which are used in defining behaviors or communications. `Actor` is Akka `Actor`, which is an interface. `Behavior` has all methods defined in `CSL`. `Handler` and `Outlet` realize the Akka `Actor`.

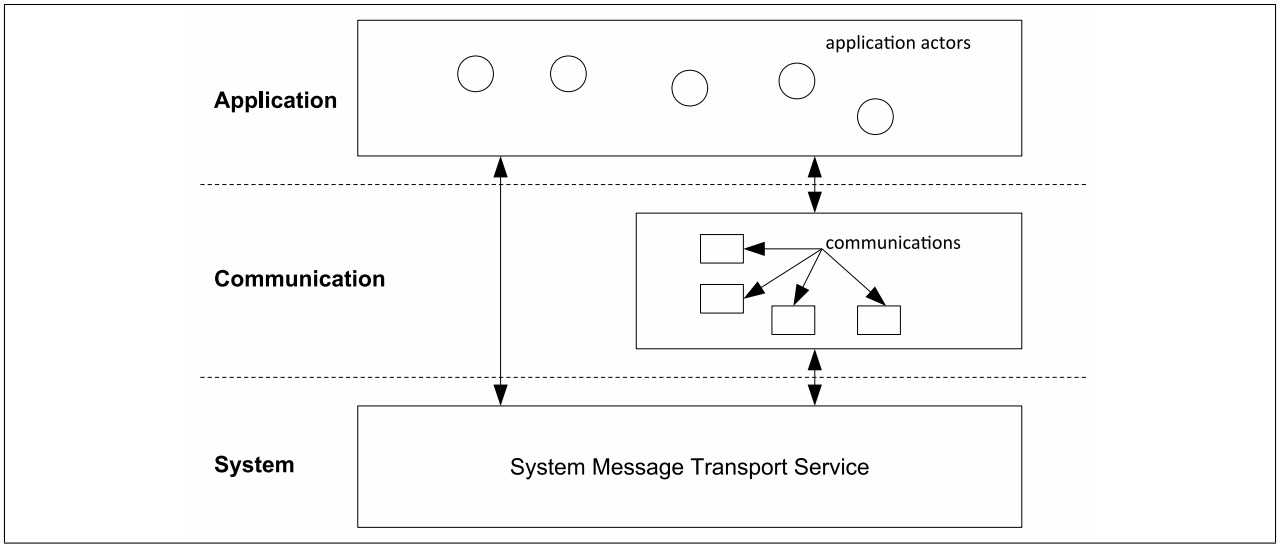

Figure 5.1: System Architecture

`Communication` inherits methods from `CSL`. Furthermore, from the class diagram, we can also see handlers and outlets have behaviors and communications have outlets and handlers.

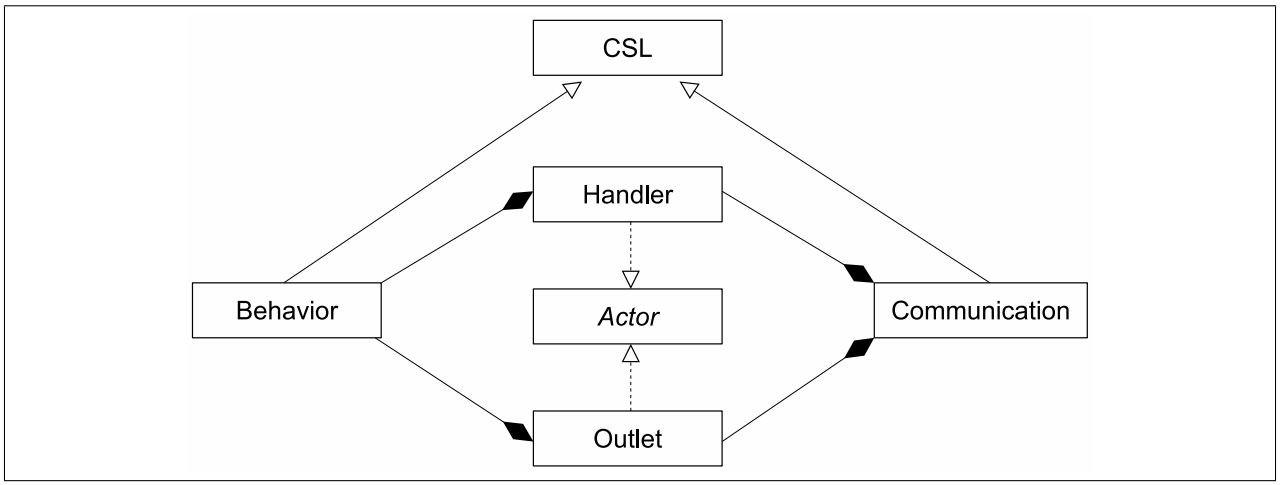

Figure 5.2: interActors Class Diagram

5.2.1 CSL

Class `CSL`, which is the superclass of `Behavior` and `Communication`, implements methods that can be used in these two classes. Because the code is too long, I list it in Section A.1 of Appendix A. The class is named after Communication Specification Language (CSL), which is defined in Chapter 6 and intends to restrict arbitrary computations in communications. All methods in `CSL` are defined in CSL, which can be used to define behaviors and communications. For more details on these methods, please refer to Chapter 6.

5.2.2 Behavior

`Behavior` is an abstract class and is defined in Figure 5.3. Each behavior runs in some context `context`, an instance of `ActorContext`. Further, each behavior has a list of recipients `targets`, which is a list of `ActorRef`, a `receive` method which is an abstract method and is executed when a message arrives at an outlet or a handler which holds the behavior, and a `setTargets` method used to change the targets of the behavior. `ActorRef` is used to refer to an actor. `msg`, which represents an incoming message, has the type of `Any` that is the super type of any Scala type. `Any` is similar to Java's super class *Object*. Once the processing of an incoming message completes, a message (i.e., the result of processing the incoming message) will be sent to the recipients if necessary. Programmers define their own concrete behavior classes by extending `Behavior` and implementing the abstract method `receive`.

```
1 abstract class Behavior(val context: ActorContext, var targets: List[ActorRef]) extends CSL {
2   def receive(msg: Any, agent: ActorRef)
3   def setTargets(ts: List[ActorRef]) = {
4     targets = ts
5   }
6 }
```

Figure 5.3: Abstract Behavior Class

Next, I show the implementations of behaviors described in Chapter 3.

Forwarder

Figure 5.4 shows the code for *forwarder*, which is defined as `Forwarder`. Upon receiving a message, a forwarder forwards it to its target `targets`. In the code, `sendm` defined in `CSL` sends message `msg` to a list of recipients `targets`.

```
1 class Forwarder(val ct: ActorContext, val t: List[ActorRef]) extends Behavior(ct, t) {
2   def receive(msg: Any, agent: ActorRef) = {
3      sendm(targets, msg)
4   }
5 }
```

Figure 5.4: Behavior: Forwarder

Counter

Figure 5.5 shows the code for *counter*, which counts the number of received messages and sends the result to its targets. In the code, `Counter` has a local variable `i`, which is incremented by 1 at the arrival of a message.

```
1 class Counter(val ct: ActorContext, val t: List[ActorRef]) extends Behavior(ct, t) {
2   var i = 1
3   def receive(msg: Any, agent: ActorRef) = {
4      sendm(targets, i)
5      i = i + 1
6   }
7 }
```

Figure 5.5: Behavior: Counter

Timer

Figure 5.6 shows the code for *timer*, which delays an incoming message for defined time and then sends the message to its targets. In the code `delay` is defined in `CSL` and is used to postpone the execution for defined `msecs` milliseconds.

```
1 class Timer(val ct: ActorContext, val t: List[ActorRef], val msecs: Int) extends Behavior(ct, t) {
2   def receive(msg: Any, agent: ActorRef) = {
3      delay(msecs)
4      sendm(targets, msg)
5   }
6 }
```

Figure 5.6: Behavior: Timer

Applier

The code for *applier* is shown in Figure 5.7. An applier applies a programmer-provided function `f` to received messages and then sends the result to its targets. Function `f` has the type of `Any => Any`.

```
1 class Applier(val ct: ActorContext, val t: List[ActorRef], val f: Any => Any) extends Behavior(ct, t) {
2   def receive(msg: Any, agent: ActorRef) = {
3      sendm(targets, f(msg))
4   }
5 }
```

Figure 5.7: Behavior: Applier

Encryption Applier can be used to encrypt received messages if the programmer-supplied function implements an encryption algorithm. Suppose `encrypt` is a function that receives a string as a parameter and returns it encrypted. Providing this function as a parameter creates an encryptor as illustrated in Figure 5.8. In the code, `context` is an instance of `ActorContext` and `targets` is a list of `ActorRef`.

`encryptor` can be used by outlets and handlers which require encrypting received messages. Similarly, *applier* can be used to decrypt received messages.

```
1 val encryptor = new Applier(context, targets, encrypt)
```

Figure 5.8: Encryptor Implemented Using Applier

Filter

The code for *filter* is shown in Figure 5.9. At the arrival of a message, a filter checks it against function `f` to determine whether it should be filtered or not. Function `f` has the type of `Any => Boolean` and is written by programmers.

```
1 class Filter(val ct: ActorContext, val t: List[ActorRef], val f: Any => Boolean) extends Behavior(ct, t) {
2   def receive(msg: Any, agent: ActorRef) = {
3      if (f(msg)) {
4         sendm(targets, msg)
5      }
6   }
7 }
```

Figure 5.9: Behavior: Filter

Selector

A selector (Figure 5.10) sends received messages to its selected targets. For each received message, it chooses a subset of recipients from its targets using the function `select` written by programmers. Therefore, for different messages, they may be sent to different targets in accordance with the definition of `select`.

```
1 class Selector(val ct: ActorContext, val t: List[ActorRef],
2              val select: (Any, List[ActorRef]) => List[ActorRef]) extends Behavior(ct, t) {
3   def receive(msg: Any, agent: ActorRef) = {
4      sendm(select(msg, targets), msg)
5   }
6 }
```

Figure 5.10: Behavior: Selector

Aggregator

Aggregator is defined as class `Aggregator` in Figure 5.11. It accepts two functions: `cond` and `aggr`, which are written by programmers. On the arrival of a message, an *aggregator* inserts it into its local message list `msgs`. Then it checks whether it stops receiving messages by calling function `cond`, which takes the message list as the input and returns a Boolean value. `cond` returns `true` if a termination condition is met, otherwise, `false`. In the case of termination, it aggregates all received messages then sends the aggregated result to its targets. The message list is reset to empty in order to be reused for next aggregation if any.

```
1 class Aggregator(val ct: ActorContext, val t: List[ActorRef],
2                  val cond: List[Any] => Boolean, val aggr: List[Any] => Any) extends Behavior(t) {
3     var msgs = List[Any]()
4    def receive(msg: Any, agent: ActorRef) = {
5       msgs = append(msg, msgs)
6       if (cond(msgs)) {
7          sendm(t, aggr(msgs))
8          msgs = List[Any]()
9       }
10   }
11 }
```

Figure 5.11: Behavior: Aggregator

5.2.3 Outlets and Handlers

Outlets and handlers are instances of class `Outlet` and class `Handler` , respectively, each of which has a behavior parameter `bhv` and extends Akka `Actor`.

Outlets

Figure 5.12 defines the class of `Outlet`, which is an actor and implements the `receive` method which is executed at the arrival of a message. It recognizes five types of messages:

- `Behv(b)` message sets the outlet with a new behavior `b`.
- "`Behavior`" returns the outlet's behavior to the querier.
- `NewTargets(t)` sets the behavior's targets to `t`.
- `Targets(t)` adds a list of new targets `t` to the behavior's existing targets.
- other message `msg` is handled by the outlet's behavior.

The first four types of message are considered as system messages, because only handlers are allowed to send such messages to outlets in order to change an outlet's behavior. The last type of messages is handled by the outlet's behavior.

```
1 class Outlet(var bhv: Behavior) extends Actor {
2   def receive = {
3     case Behv(b) => bhv = b
4     case "Behavior" => sender ! bhv
5     case NewTargets(t) => bhv.setTargets(t)
6     case Targets(t) => bhv.setTargets(bhv.getTargets ::: t)
7     case msg => bhv.receive(msg, self)
8   }
9 }
```

Figure 5.12: Outlet

Handlers

Likewise, `Handler` is an actor and implements `receive` method which is executed at the arrival of a message. Handlers do not recognize `Behv(b)`, "`Behavior`", `NewTarget(t)`, and `Targets(t)` because handlers do not change its behavior and targets.

```
class Handler(var bhv: Behavior) extends Actor {
  def receive = {
    case msg => bhv.receive(msg, self)
  }
}
```

Figure 5.13: Handler

5.2.4 Communication

All programmer-specified communication types are subclassed from `Communication` (Figure 5.14). Each communication has `attributes`, `inlets`, `outlets`, and `handlers`. `inlets`, `outlets` and `handlers` are used to keep track of input outlets, output outlets, and handlers, respectively.

```
abstract class Communication (val context: ActorContext) extends CSL {
  var attributes = Map.empty[String, Any]
  var inlets = List[ActorRef]()
  var outlets = List[ActorRef]()
  var handlers = List[ActorRef]()

  def getInlets = inlets
  def getOutlets = outlets
  def getHandlers = handlers
  def addInlets(ins: List[ActorRef]) = {
    inlets = inlets ::: ins
  }
  def addOutlets(outs: List[ActorRef]) = {
    outlets = outlets ::: outs
  }
  def addHandlers(hdlers: List[ActorRef]) = {
    handlers = handlers ::: hdlers
  }

  def setAttr(attrs: Map[String, Any]) = {
    attributes = attrs
    terminate()
    init() // initialize outlets and handlers
  }
  def terminate() = {
    inlets.foreach(context.stop(_)) // stop inlets
    inlets = List[ActorRef]()
    outlets.foreach(context.stop(_)) // stop outlets
    outlets = List[ActorRef]()
    handlers.foreach(context.stop(_)) // stop handlers
    handlers = List[ActorRef]()
  }

  def launch()
  def init()
}
```

Figure 5.14: Abstract Communication Class

Figure 5.15 shows a communication object that has one input outlet, one handler, and one output outlet. The rectangle labeled with `CO` represents the object. The three ovals labeled with `in`, `hdr`, and `out` are the

input outlet, the handler, and the output outlet, respectively. The lines from the object to ovals denote that the object has references to these actors.

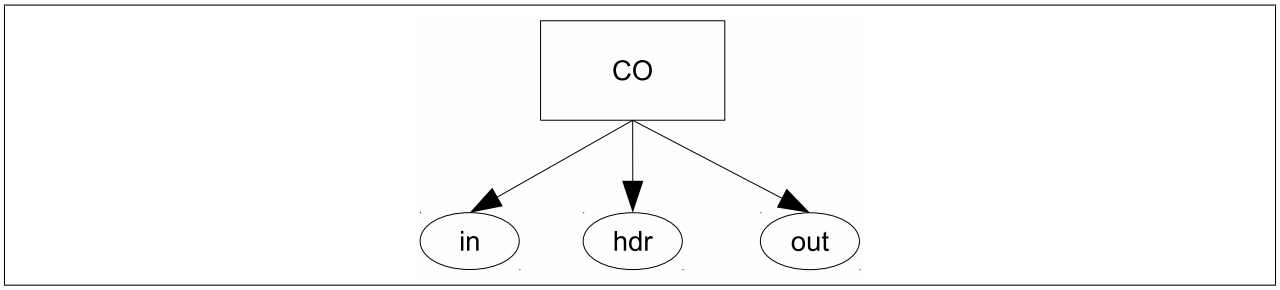

Figure 5.15: A Communication Object

Getter and Setter Methods

`getInlets`, `getOutlets`, and `getHandlers` return the input outlets, the output outlets, and the handlers, respectively, of a communication. `addInlets`, `addOutlets`, and `addHandlers` are used to add a list of input outlets `ins`, a list of output outlets `outs`, and a list of handlers `hdlers` to existing input outlets, output outlets, and handlers, respectively.

Destroy Method

The method `terminate` stops outlets and handlers if any. The method is specially used when resetting a communication's attributes which establishes initial rendezvous for another set of participants. The run-time system can call this method in order to terminate a communication.

Attribute Method

After a communication is created, `setAttr` is called to initialize its outlets and handlers. Figure 5.16 shows this case. We can see after calling `setAttr`, the outlets and the handler are created and the communication object has references to them. As a matter of fact, it is the `init` method to create outlets and handlers, which will be discussed shortly.

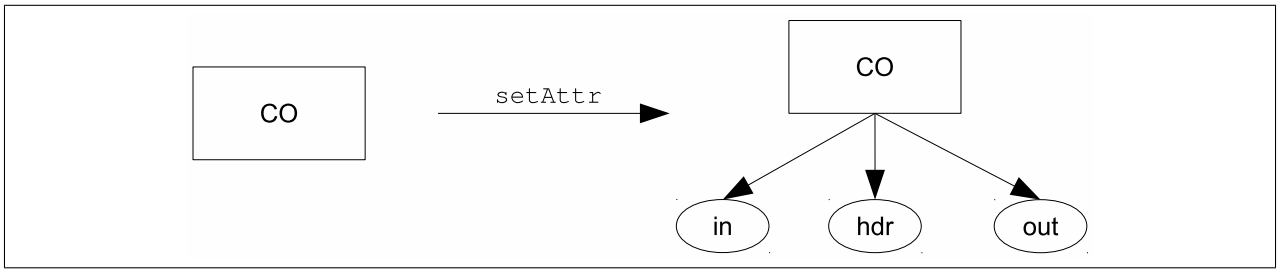

Figure 5.16: First Call to `setAttr`

A communication can be reused by reseting its attributes. This is done by invoking another call to `setAttr`. This case is illustrated in Figure 5.17. The second call resets the communication's attributes,

destroys existing outlets and handlers, and creates new outlets and handlers for the communication. Although through `setAttr`, we have a new set of outlets and handlers for a communication, the number of outlets and handlers is not changed because it is defined in the class of the communication.

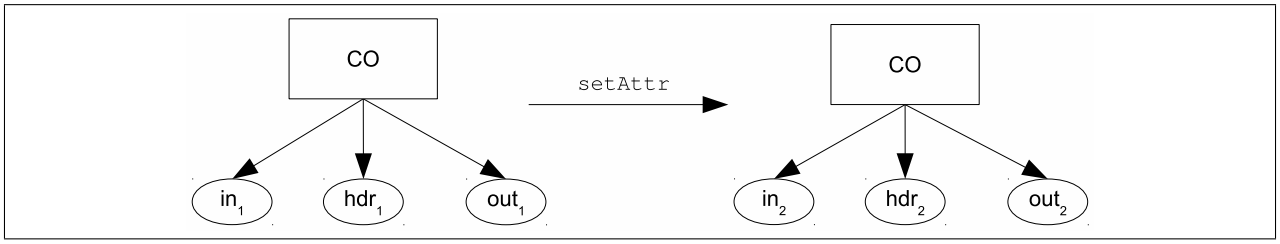

Figure 5.17: Second Call to `setAttr`

Abstract Methods

`init` and `launch` are both abstract methods, which must be implemented by a concrete communication class. `init` initializes (or creates if they do not exist) the outlets and handlers for a communication. The number of outlets and handlers of a communication type has is defined in a concrete subclass of `Communication` class.

A communication is launched by calling `launch`. Typically, `launch` sends a communications participants the relevant names of its input outlets. Figure 5.18 shows this scenario. Suppose there are three participants and the communication has one input outlet. On the left side, the communication `CO` has an input outlet in_1 and the three participants do not know the existence of the input outlet. After invoking `launch`, the name of the input outlet is sent to the participants. In this figure, the three lines out from in_1 means the name of in_1 is sent to the participants. Once the participants know the input outlet, they can send messages to it.

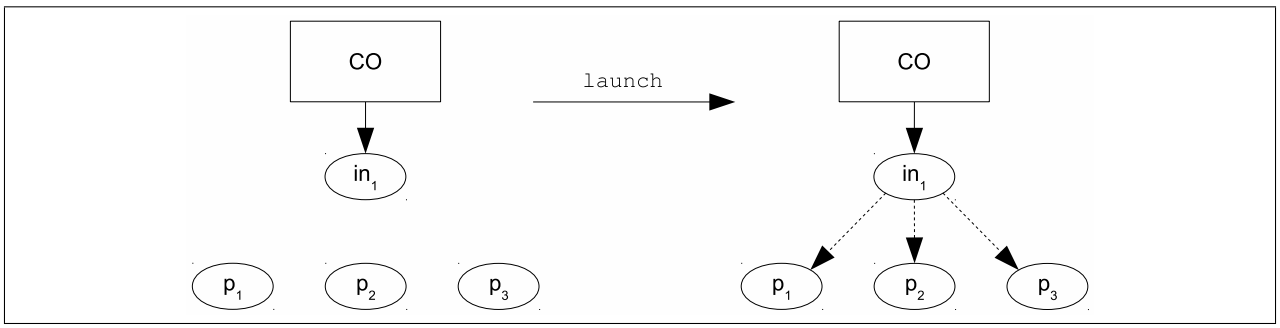

Figure 5.18: Call to `launch`

5.2.5 The Life Cycle of a Communication

Figure 5.19 shows the life cycle of a communication object, which includes four states: *new*, *initialized*, *running*, and *terminated*.

A communication comes into existence once an application creates it. At this stage, although a communication exists as an object, its attributes are not yet set and its outlets and handlers do not exist yet. Its

state becomes *initialized* through the initialization operation. Launching operation makes the communication *running*. For some long-lived communications, the *running* state is their final state. For others that should terminate, the final state is *terminated*, in which a communication may be garbage collected.

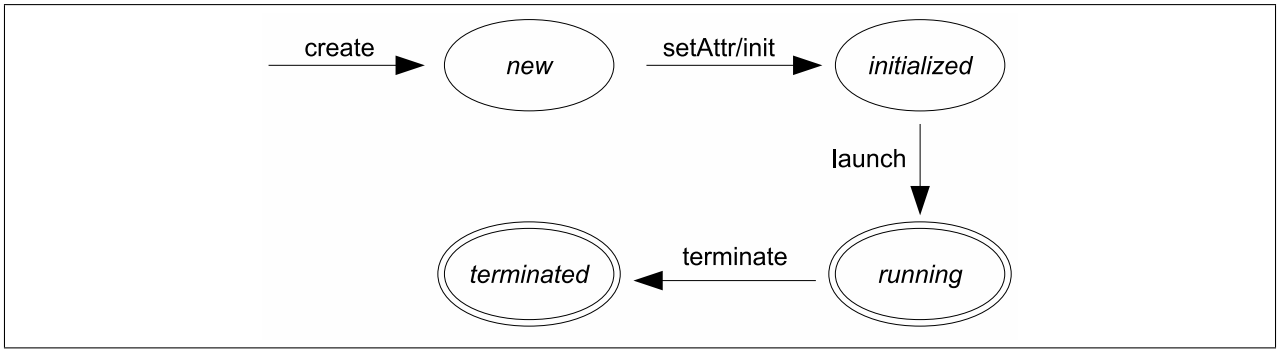

Figure 5.19: The Life Cycle of a Communication

5.3 Examples: Communications

To show how to implement a communication, this section presents three communication examples: *broadcaster*, *router*, and *Multi-Origin Many-to-Many*.

5.3.1 Broadcaster

A *broadcaster* communication shown in Figure 5.20 has one input outlet *in* and one output outlet *out*. Both outlets have the behavior of *forwarder*. Any actors (represented by ovals) connected to the input outlet of *broadcaster* can use it to broadcast a message to actors connected to the output outlet. Particularly, in the figure, three actors know the name of the input outlet, and three actors are connected to the output outlet. The input outlet forwards any received message to the output outlet, which, in turn, forwards the message to all connected actors.

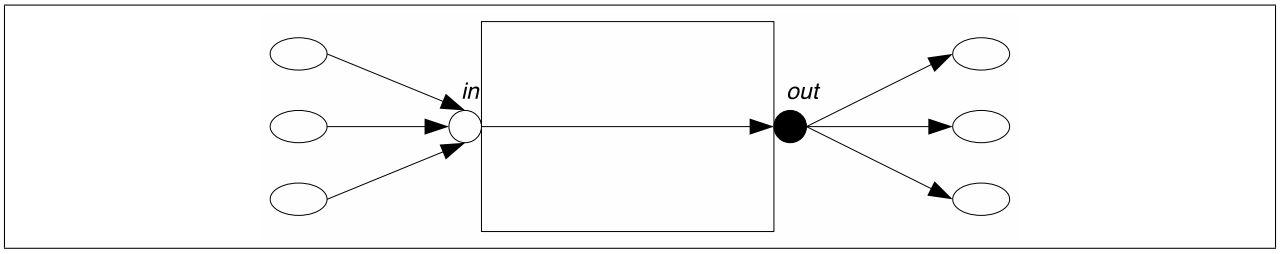

Figure 5.20: Communication: Broadcaster

Figure 5.21 shows the code. Its output outlet is created with a behavior of `Forwarder` and thus forwards received messages to the communication's targets. Likewise, its input outlet forwards messages to the output outlet. `init` method retrieves attributes which is set by `setAttr`, and create the output outlet and the input outlet. In line 10 and line 12, `context` is an instance of `ActorContext` and is used to create an

actor in `createOutlet` method, which is defined in `CSL`. `createOutlet` creates an input outlet if the third parameter is `input` or an output outlet if the third parameter is `output`. `launch` method launches the communication, in which it tells all participants the name of the input outlet so that participants can send it messages. On the arrival of a message from a participant at the input outlet, the input outlet sends the message to the output outlet because the former targets the latter. Finally, the output outlet forwards the message to the recipients of the communication.

```
1 class Broadcaster(ct: ActorContext) extends Communication(ct) {
2   var out: ActorRef = null
3   var in: ActorRef = null
4   var recipients: List[ActorRef] = null
5   var participants: List[ActorRef] = null
6   def init() = {
7     recipients = attr(attributes, "recipients").asInstanceOf[List[ActorRef]]
8     participants = attr(attributes, "participants").asInstanceOf[List[ActorRef]]
9     // output outlet
10    out = createOutlet(context, this, "output", new Forwarder(context, recipients))
11    // input outlet
12    in = createOutlet(context, this, "input", new Forwarder(context, List(out)))
13
14  }
15  def launch() = {
16    sendm(participants, in)
17  }
18 }
```

Figure 5.21: Code: Broadcaster Communication

Broadcaster communication decouples senders and recipients so that they are not required to know each other in order to communicate. It can be seen as a simple implementation of publisher-subscriber communication pattern [29]. We call it simple because message senders and message recipients do not distinguish topics, whereas, publisher-subscriber pattern does. Now, suppose we have a different behavior that implements logics for discerning topics. If we use this behavior for either the input outlet or the output outlet, we get a general-purpose publisher-subscriber system.

5.3.2 Router

Figure 5.22 shows a *router* communication, which has one input outlet *in*, one handler *selector*, and two output outlets out_1 and out_2. To simplify the figure, I do not draw actors. Note that each outlet can be connected to multiple actors. The handler has the behavior of *selector*, which sends messages to either of the two output outlets but not both at the same time.

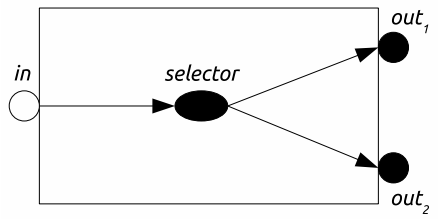

Figure 5.22: Communication: Router

The code for *router* is presented in Figure 5.23. `Router` has recipients `recipients`, participants `parts`, and a function `select` used to instantiate the selector behavior, one input outlet `inlet`, one handler `handler`, and two output outlets: `outlet1` and `outlet2`. All outlets have the behavior of *forwarder*. The handler has the behavior of *selector* and is created by `createHandler`, which is defined in class `CSL`. At the time of launching, *router* notifies all participants the name of the input outlet. In the code, we assume there are two recipients in the recipient list `recipients`. `outlet1` has the first recipient as its target, and `outlet2` has the second one as its target. When a message from a participant arrives at the input outlet, it is forwarded to the handler that is a *selector*. The selector sends the message to one of the two output outlets by checking the message with the programmer-provided function `select`. The output outlet that receives the message forwards it to its targets.

```
1 class Router(ct: ActorContext) extends Communication(ct) {
2   var recipients: List[ActorRef] = null;
3   var parts: List[ActorRef] = null
4   var select: (Any, List[ActorRef]) => List[ActorRef] = null;
5
6   var inlet: ActorRef = null
7   var handler: ActorRef = null
8   var outlet1: ActorRef = null
9   var outlet2: ActorRef = null
10
11  def launch() = {
12    sendm(parts, inlet);
13  }
14
15  def init() = {
16    recipients = attr(attributes, "recipients").asInstanceOf[List[ActorRef]]
17    parts = attr(attributes, "parts").asInstanceOf[List[ActorRef]]
18    select = attr(attributes, "select").asInstanceOf[(Any, List[ActorRef]) => List[ActorRef]]
19
20    // create outlets and the handler
21    outlet1 = createOutlet(context, this, "output", new Forwarder(context, List(recipients(0))))
22    outlet2 = createOutlet(context, this, "output", new Forwarder(context, List(recipients(1))))
23    handler = createHandler(context, this, new Selector(context, List(outlet1, outlet2), select))
24    inlet = createOutlet(context, this, "input", new Forwarder(context, List(handler)))
25  }
26 }
```

Figure 5.23: Code: Router Communication

This implementation has a limitation. That is, *router* has only two output outlets. As a consequence, it only chooses one of them as a recipient. If an application requires more than two targets, this version of implementation is not able to handle it. I will discuss another version of implementation in Section 5.6.

5.3.3 Multi-Origin Many-to-Many

Multi-Origin Many-to-Many (MOM2M) communication shown in Figure 5.24 has one input outlet and one output outlet. From the surface, it is the same as a channel. However, it is different because we make its output outlet have *aggregator* behavior defined in Figure 5.11, whereas, the output outlet of a channel has the behavior of *forwarder*.

`Mom2m` (Figure 5.25) has a set of recipients `recipients`, a group of participants `participants`, condition function `cond`, aggregation function `aggr`, one input outlet `inlet` and one output outlet `outlet`.

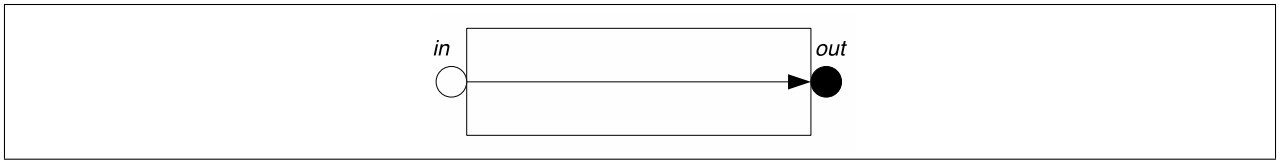

Figure 5.24: Communication: Multi-Origin Many-to-Many

The input outlet has *forwarder* behavior and the output outlet has *aggregator* behavior. For an MOM2M communication, incoming messages arrive at its input outlet, which, in turn, forwards the messages to its output outlet. Because the output outlet has the behavior of *aggregator*, the received messages are aggregated once the pre-defined condition is satisfied checked by `cond`. The aggregated result is sent to the recipients of the communication.

By passing different condition function `cond` and an aggregation function `aggr`, programmers can accomplish different tasks using an MOM2M. For example, an MOM2M can be used as a barrier to synchronize multiple concurrent computations: suppose there are n computations. Programmers can provide a `cond` function which returns `true` when n messages have been received, one from each of the computations.

```
1  class Mom2m(ct: ActorContext) extends Communication(ct) {
2    var recipients: List[ActorRef] = null;
3    var participants: List[ActorRef] = null;
4    var cond: List[Any] => Boolean = null
5    var aggr: List[Any] => Any = null
6
7    var outlet: ActorRef = null
8    var inlet: ActorRef = null
9
10   def launch() = {
11     sendm(participants, inlet);
12   }
13   def init() = {
14     recipients = attr(attributes, "recipients").asInstanceOf[List[ActorRef]]
15     participants = attr(attributes, "participants").asInstanceOf[List[ActorRef]]
16     cond = attr(attributes, "cond").asInstanceOf[List[Any] => Boolean]
17     aggr = attr(attributes, "aggr").asInstanceOf[List[Any] => Any]
18
19     // create output outlet
20     val aggregator = new Aggregator(context, recipients, cond, aggr)
21     outlet = createOutlet(context, this, "output", aggregator)
22
23     // create input outlet
24     val input = new Forwarder(context, List(outlet))
25     inlet = createOutlet(context, this, "input", input)
26   }
27 }
```

Figure 5.25: Code: MOM2M Communication

Note that the behaviors of the output outlet and the input outlet can be switched, because all we need is one of the two outlets doing the aggregation. In fact, our implementation allows a single outlet to act for both an input and an output outlet, allowing MOM2M to be implemented with only one outlet.

Another implementation of MOM2M communication can be making it have one input outlet, one handler, and one output outlet as shown in Figure 5.26. In this implementation, aggregation is separated from the output outlet and is carried out by the handler. The code for this implementation is shown in Section A.2 in Appendix A.

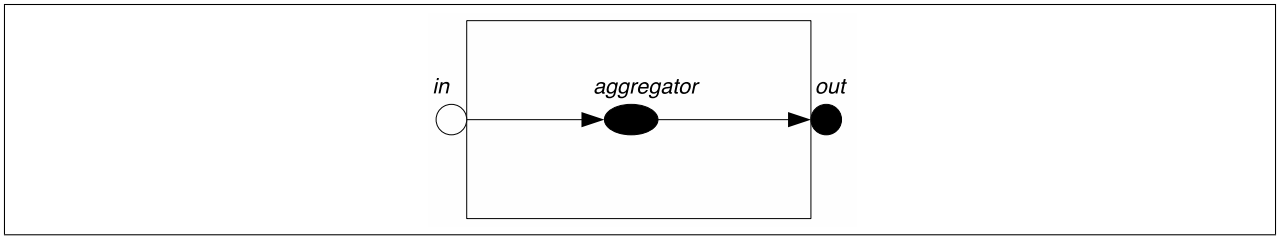

Figure 5.26: Communication: MOM2M Version 2

The key difference between MOM2M and other aggregation communication protocols, such as message continuation [79], activators [32], reduction communication pattern [21], and join continuation [77] is that the former does not require every participant to respond because of the existence of the termination condition function.

5.4 Communication-Oriented Programming

This section presents Communication-Oriented Programming (COP). Generally speaking, COP requires the following three steps:

- Implement required behaviors if they do not exist in the behavior library
- Implement the communications used in the application if they are not available in the communication library
- Use the communications

I illustrate each step by walking through two examples – SOM2M presented in Chapter 1 and barrier implemented using MOM2M.

5.4.1 Example: Single-Origin Many-to-Many

The SOM2M communication has the same pictorial view as MOM2M shown in Figure 5.26, and it has one input outlet, one handler, and one output outlet. However, the behavior of the handler and the behavior of the output outlet are different from their counterparts in MOM2M. The input outlet receives *votes* from participants and forwards them to the handler, the handler determines whether sufficient number of responses has received and if so, aggregates them into a boolean value and sends the value to the output outlet. The output outlet holds the draft message. Once it has received a message from the handler, it decides whether the draft message should be sent to the recipients.

Implementing Behaviors

Through the above analysis, the input outlet has the behavior of *forwarder*, which is defined in Figure 5.4. The handler has the behavior similar to *aggregator* defined in Figure 5.11, but it slightly different. In SOM2M,

the handler only needs to do aggregation once and sends a message to the output outlet once. In other words, the aggregator does not need to be reused. The output outlet has the behavior that can hold the draft message using a local variable and sends it if it receives a `true` value. We define the two behaviors in Figure 5.27 and Figure 5.28, respectively.

Compared to `Aggregator`, `AggregatorOneOff` has one more local variable `sent` indicating whether it has already sent a message. If yes, it does nothing on an incoming message; otherwise, it adds the incoming message to its local message list, checks whether it stops receiving messages, sends out an aggregated message if yes. Because `AggregatorOneOff` just does one-time aggregation, its local message list is not reset.

```
1 class AggregatorOneOff(val ct: ActorContext, val t: List[ActorRef],
2                  val cond: List[Any] => Boolean, val aggr: List[Any] => Boolean) extends Behavior(t) {
3     var msgs = List[Any]()
4     var sent = false;
5    def receive(msg: Any, agent: ActorRef) = {
6       if (sent == false) {
7          msgs = append(msg, msgs)
8          if (cond(msgs)) {
9             sendm(t, aggr(msgs))
10            sent = true
11         }
12      }
13   }
14 }
```

Figure 5.27: Code: One-Off Aggregator Behavior

In addition to running context and a list of recipients, `SpoutOneOff` behavior has `content` to hold a value. The behavior only processes messages with the type of Boolean. If a message has the other types, the behavior does nothing. Once it receives a message which has the value of `true`, it sends the value of `content` to targets. Like `AggregatorOneOff`, `SpoutOneOff` use a local variable `sent` to determine whether a message is sent. That is, `SpoutOneOff` only sends out its content once.

```
1 class SpoutOneOff(val ct: ActorContext, val t: List[ActorRef],  val content: Any) extends Behavior(t) {
2    var sent = false
3    def receive(msg: Any, agent: ActorRef) = {
4       if (sent == false) {
5          if (msg.isInstanceOf[Boolean]) {
6             if (msg.asInstanceOf[Boolean] == true) {
7                sendm(targets, content)
8                sent = true
9             }
10         }
11      }
12   }
13 }
```

Figure 5.28: Code: One-Off Spout Behavior

Implementing Communications

SOM2M communication is defined in Figure 5.29 as `Som2m`, which has six attributes: a list of targets `recipients`, a message payload `content` to store the draft message, a query message `query`, a list of participants `parts`, condition function `cond`, and aggregation function `aggr`. The input outlet `in` has

the behavior of `Forwarder`, the output outlet `out` has `SpoutOneOff` behavior, and the handler has the behavior of `AggregatorOneOff`. At the time of launching, participants are notified about the name of the input outlet.

```
1  class Som2m(ct: ActorContext) extends Communication(ct) {
2    var recipients: List[ActorRef] = null
3    var content: Any = null
4    var query: Any = null
5    var parts: List[ActorRef] = null
6    var cond: List[Any] => Boolean = null
7    var aggr: List[Any] => Boolean = null
8
9    var out: ActorRef = null
10   var aggregator: ActorRef = null
11   var in: ActorRef = null
12   def init() = {
13     // retrieve attributes
14     recipients = attr(attributes, "recipients").asInstanceOf[List[ActorRef]]
15     content = attr(attributes, "content").asInstanceOf[Any]
16     query = attr(attributes, "query").asInstanceOf[Any]
17     parts = attr(attributes, "parts").asInstanceOf[List[ActorRef]]
18     cond = attr(attributes, "cond").asInstanceOf[List[Any] => Boolean]
19     aggr = attr(attributes, "aggr").asInstanceOf[List[Any] => Boolean]
20
21     // create the output outlet, the handler, and the input outlet
22     out = createOutlet(context, this, "output", new SpoutOneOff(context, recipients, content))
23     aggregator = createHandler(context, this, new AggregatorOneOff(context, List(out), cond, aggr))
24     in = createOutlet(context, this, "input", new Forwarder(context, List(aggregator)))
25   }
26   def launch() = {
27     tellm(in, parts, query);
28   }
29 }
```

Figure 5.29: Single-Origin Many-to-Many Communication

Using Communications

The steps of using a communication in an application including creating an instance of the communication, initiating the communication instance by setting its attributes, and launching the communication.

Figure 5.30 demonstrates the usage of SOM2M. Suppose there are `n` participants and the number of required "yes" votes is `required`. `r` is the message recipients and `parts` holds a list of participants. `cond` and `aggr` functions are defined in the object `funs`. `cond` returns `true` when either of two conditions is satisfied: (1) the number of agreements is greater than or equal to `required`, (2) the number of disagreement plus `required` is greater than total number of participants. Once `cond` returns `true`, the aggregation process starts by calling `aggr`, which finds the number of agreements in the list and returns `true` if it is greater than or equal to `required`, `false` otherwise.

The three steps of using a communication are shown in line 5, line 6, and line 13, respectively. The complete code is shown in Appendix A Section A.3.

In this section, we use SOM2M to show how to develop an application using communications. In SOM2M, the output outlet only accepts a Boolean value but messages having other types are discarded. As application requirements change, we can easily change the behavior of the output outlet and the implementation of `aggr` function to accommodate new requirements. For example, in current SOM2M case, if the number of

```
var cond = funs.cond(n, required, _: List[Any])
var aggr = funs.aggr(required, _: List[Any])
val payload = "some meaningful information"

val som2m = new Som2m(context)
som2m.setAttr(Map(
        "recipients" -> List(r),
        "content" -> payload,
        "query" -> "vote this:" + payload,
        "parts" -> parts,
        "cond" -> cond,
        "aggr" -> aggr))
som2m.launch()

object funs {
  def cond(total: Int, required: Int, l: List[Any]): Boolean = {
    var na = 0 // no. of agreement
    var nd = 0 // no. of disagreement
    // go through the message list to get na and nd
    l.foreach(x =>
      if (x == "yes") {
        na = na + 1
      } else {
        nd = nd + 1
      })
    if (na >= required || (nd + required) > total)
      true
    else
      false
  }

  def aggr(required: Int, l: List[Any]): Boolean = {
    var na = 0 // no. of agreement
    l.asInstanceOf[List[String]].foreach(x => if (x == "yes") na = na + 1)
    if (na >= required) true else false
  }
}
```

Figure 5.30: Using SOM2M Communication

agreement is less than the required number, no message will be sent to the recipient and no participant is notified. Consider the application should send a failure (for instance) message to the recipient and notify all participants a failure message, too. We can change the behavior of the output outlet at runtime to meet these requirements.

5.4.2 Example: Barriers

We use MOM2M communication to implement a barrier. Since the communication itself and behaviors used in the communication have already defined previously, in this example, I only show how to use this communication.

Figure 5.31 shows the code for implementing a barrier using MOM2M. In this implementation, the recipients and participants of the MOM2M communication are the same set of processes `processes`. Suppose there are n processes. Functions `cond` and `aggr` are defined in the object `funs`. On the arrival of each message from a process, `cond` checks whether the number of messages received has reached n. If so, it returns `true`, triggering aggregation, which in this case just returns a "go" message, which is sent to all the processes. Once a process receives a "go" message, it starts a new iteration if any. The entire code for implementing a barrier and testing it is listed in Section A.4 in Appendix A. From the code, we can see the implementation of a barrier using `Mom2m` only requires defining two functions: `cond` and `aggr`.

```
var cond = funs.cond(n, _: List[Any])
var aggr = funs.aggr(_:List[Any])

// create a Mom2m, initialize it, and launch it
val barrier = new Mom2m(context)
barrier.setAttr(Map("recipients" -> processes, "participants" -> processes, "cond" -> cond, "aggr" -> aggr))
barrier.launch()

// functions used to initialize a barrier
object funs {
  def cond(total: Int, l: List[Any]): Boolean = {
    if (l.length == total)
      true
    else
      false
  }

  def aggr(l: List[Any]): Any = {
     "go"
  }
}
```

Figure 5.31: Barrier Implementation Using MOM2M

5.5 Composition

I have implemented the three ways of composing communications: input merge, output merge, and output-input merge. In our implementation, I use a dummy communication to hold the composed communication. The definition of `Dummy` is shown in Figure 5.32, which does not have any outlets and handlers, and the two methods `init` and `launch` have the empty body.

```
class Dummy(ct: ActorContext) extends Communication(ct) {
  def launch() = {}
  def init() = {}
}
```

Figure 5.32: Dummy Communication

Input Merge

Input merge is implemented as method `Inputs` (Figure 5.33), which takes an actor context, a list of communications `coms`, a list of input outlets to be merged `ins`, and a behavior `b` used by the newly created handler.

In the code, we first create a dummy communication used as a container of the composed communication. Then, for the dummy communication, we create a handler with behavior `b` targeting `ins`. Next, we add all existing handlers of communications `coms` to the composed communication. Then, we add all input outlets but the merged ones and all output outlets to the composed communication. Finally, the composed communication is returned.

```
def Inputs(context: ActorContext, coms: List[Communication], ins: List[ActorRef], b: Behavior) = {
  // create a dummy communication
  val cc = new Dummy(context)

  // create handler with behavior b, which targets ins
  b.setTargets(ins);
  val handler = createHandler(context, cc, b)
  // add all handlers of all communications to the composed communication
  var handlers = List[ActorRef]()
  coms.foreach(c => handlers = handlers ::: c.getHandlers)
  cc.addHandlers(handlers)

  // create input outlet with behavior Forwarder
  val input = new Forwarder(context, List(handler))
  val inlet = createOutlet(context, cc, "input", input)

  // get all input outlets
  var inlets = List[ActorRef]()
  coms.foreach(c => inlets = inlets ::: c.getInlets)
  // remove merged input outlets
  inlets = inlets.filterNot(ins.contains(_))
  // add input outlets to the composed communication
  cc.addInlets(inlets)

  // add output outlets to the composed communication
  var outlets = List[ActorRef]()
  coms.foreach(c => outlets = outlets ::: c.getOutlets)
  cc.addOutlets(outlets)

  cc
}
```

Figure 5.33: Code: Input Merge

Output Merge

Output merge is implemented as method `Outputs` (Figure 5.34), which takes an actor context, a list of communications `coms`, a list of output outlets to be merged `outs`, a behavior `b` used by the newly created handler, and a new target `t`.

In the code, we first create a dummy communication as a holder of the composed communication. Then, we add all input outlets to the composed communication. Next, we create an output outlet having behavior `Forwarder` targeting `t` for the dummy communication. Then, we add all existing output outlets but merged ones to the composed communication. After we create a handler with behavior `b` targeting the newly created output outlet, we add all existing handlers of communications `coms` to the composed communication. Finally, we change the merged output outlets to target the newly created handler and return the composed communication.

Output Input Merge

`OutIn` (Figure 5.35) implements output-input merge, which connects output outlets to input outlets. The parameters it takes are an actor context `context`, a list of communications `coms`, and a list of bindings `bindings`, each of which is a 3-tuple having the form of (*output_outlet, input_outlet, behavior*).

The composition process is as follows: (1) create a dummy communication `cc`, which is the composed communication. (2) create a handler for each tuple in `bindings` with the provided behavior targeting the input outlet in the same tuple. (3) add existing handlers to the composed communication. (4) add

```
def Outputs(context: ActorContext, coms: List[Communication], outs: List[ActorRef],
               b: Behavior, t: List[ActorRef]) = {
  // create a dummy communication
  val cc = new Dummy(context)

  var inlets = List[ActorRef]()
  coms.foreach(c => inlets = inlets ::: c.getInlets)
  cc.addInlets(inlets)

  // create output outlet with behavior Forwarder and target t
  val output = new Forwarder(context, t)
  val outlet = createOutlet(context, cc, "output", output)
  // add all output outlets but merged output outlets to the composed communication
  var outlets = List[ActorRef]()
  coms.foreach(c => outlets = outlets ::: c.getOutlets)
  outlets = outlets.filterNot(outs.contains(_))
  cc.addOutlets(outlets)

  // set b's targets to newly created outlet and create handler with behavior b
  b.setTargets(List(outlet));
  val handler = createHandler(context, cc, b)
  // add existing handlers of composing communications to the composed communication
  var handlers = List[ActorRef]()
  coms.foreach(c => handlers = handlers ::: c.getHandlers)
  cc.addHandlers(handlers)

  // change the target of composed outlets
  outs.foreach(_ ! Targets(List(handler)))

  cc
}
```

Figure 5.34: Code: Output Merge

existing outlets but merged outlets in the bindings to the composed communication. (5) return the composed communication.

Example

Now, we use input merge as an example to illustrate how to compose communications. The code for output merge and output-input merge are listed in Appendix A Section A.5 and Section A.6, respectively.

Particularly, we compose three communications as shown in Figure 4.7. The composed communication defined as `InputMerge` shown in Figure 5.36 are built from three `Broadcaster` communications, each of them has its own recipients: `recipients1`, `recipients2`, and `recipients3`, respectively. `senders` are those computations that use `InputMerge` to send messages. The three `Broadcaster` communications are composed at their input outlets: `List(c1.in, c2.in, c3.in)`, and the behavior of newly created handler for merging is *forwarder*. Note that one can use different behaviors, such as *applier*, *filter*, etc. The `Inputs` method is where the composition occurs and returns a composed communication `cc`. Then, all outlets and handlers of returned communication are added to the `InputMerge` communication. Finally, the communication informs senders the name of its input outlet. Because there is only one input outlet for the composed communication, the code use `inlets(0)` to get it.

```
def OutIn(context: ActorContext, coms: List[Communication], bindings: List[(ActorRef, ActorRef, Behavior)]) = {
  var ins = List[ActorRef]()
  var outs = List[ActorRef]()
  val cc = new Dummy(context)

  bindings.foreach(b => {
    // set the target of the handler's behavior to the input outlet
    b._3.setTargets(List(b._2))
    val handler = createHandler(context, cc, b._3)
    b._1 ! Targets(List(handler))
    // merged input outlets
    ins = ins ::: List(b._2)
    // merged output outlets
    outs = outs ::: List(b._1)
  })

  // add all handlers of all communications
  var handlers = List[ActorRef]()
  coms.foreach(c => handlers = handlers ::: c.getHandlers)
  cc.addHandlers(handlers)

  // add all input outlets but merged input outlets in the bindings
  var inlets = List[ActorRef]()
  coms.foreach(c => inlets = inlets ::: c.getInlets)
  inlets = inlets.filterNot(ins.contains(_))
  cc.addInlets(inlets)

  // add all output outlets but merged output outlets in the bindings
  var outlets = List[ActorRef]()
  coms.foreach(c => outlets = outlets ::: c.getOutlets)
  outlets = outlets.filterNot(outs.contains(_))
  cc.addOutlets(outlets)

  cc
}
```

Figure 5.35: Code: Ouput-Input Merge

5.6 Summary

This chapter presents a prototype implementation of *interActors*. We assume there is a library of behaviors that can be used by outlets and handlers, and a library of communications that can be employed by applications.

In the case of a behavior does not exist in the library, programmers should develop it. Then the behavior can be used to define a communication. To develop an application based on communications, the communications are either available in existing library or developed if not available. Once programmers have a communication class, they can instantiate a communication from it, set attributes for the communication, and launch it.

Although any Scala statement is allowed in the body of a behavior definition and a communication definition for now, we do not intend to permit arbitrary code in the definitions. I discuss my effort to restrict arbitrarily complex code in the definitions of behaviors and communications in Chapter 6.

Although in Chapter 3, I mentioned that outlets are used to receive messages from and send messages to actors, and handlers process communication logic, in our implementation, we allow outlets and handlers to have the same set of behaviors. Consequently, outlets can have the same message handling capacity as handlers. Therefore, there is an opportunity to optimize our implementation by reducing the number of outlets and handlers in a communication. In this chapter, I implement a number of communications and

```
class InputMerge(ct: ActorContext) extends Communication(ct) {
  val c1 = new Broadcaster(ct)
  val c2 = new Broadcaster(ct)
  val c3 = new Broadcaster(ct)

  var recipients1: List[ActorRef] = null
  var recipients2: List[ActorRef] = null
  var recipients3: List[ActorRef] = null
  var senders: List[ActorRef] = null

  def init() = {
    recipients1 = attr(attributes, "recipients1").asInstanceOf[List[ActorRef]]
    recipients2 = attr(attributes, "recipients2").asInstanceOf[List[ActorRef]]
    recipients3 = attr(attributes, "recipients3").asInstanceOf[List[ActorRef]]
    senders = attr(attributes, "senders").asInstanceOf[List[ActorRef]]
  }
  def launch() = {
    c1.setAttr(Map("recipients" -> recipients1, "participants" -> null))
    c2.setAttr(Map("recipients" -> recipients2, "participants" -> null))
    c3.setAttr(Map("recipients" -> recipients3, "participants" -> null))

    // a list of outlets to be merged
    val inset = List(c1.in, c2.in, c3.in)

    // behavior for newly created handler
    val b = new Forwarder(ct, inset)

    // merge
    val cc = Inputs(context, List(c1, c2, c3), inset, b)

    // add existing input outlets, output outlets, and handlers
    addInlets(cc.getInlets)
    addHandlers(cc.getHandlers)
    addOutlets(cc.getOutlets)

    // notify computations which use this communication
    val inlets = getInlets
    sendm(senders, inlets(0))
  }
}
```

Figure 5.36: Example: Input Merge Three Broadcasters

each of them has input outlets and output outlets. Based on the above discussions, those implementation can be optimized by using only one outlet, which can play the roles of input outlet, handler, and output outlet. For example, for the MOM2M communication, we can remove the input outlet and allow participants to send messages to the output outlet directly. Therefore, for this example, conceptually, there are three entities in a communication: the input outlet, the handler, the output outlet. However, in implementation, there is only one entity which plays the three roles altogether.

In some circumstances, reducing the number of outlets and handlers may improve the scalability of a communication. For instance, existing *router* communication has two output outlets, which restricts the communication can only send messages to one of two sets of recipients. What if there are more sets of recipients? Alternatively, *router* can be implemented using one outlet, which receives messages from actors, selects a set of actors as targets, and sends messages to selected actors. Each time, at the receipt of a message, a different set of actors may be chosen as new targets. The code for this implementation is illustrated in Section A.7 in Appendix A.

Chapter 6

Communication Specification Language

Although communications can be coded using our implementation directly in Scala as shown in Chapter 5, I have also developed a special language called Communication Specification Language (CSL) to specify communications. The intention is to restrict arbitrary computations from being included in communications. In other words, CSL only allows specification of communications which can be constructed using the composition rules presented in Chapter 3. The purpose of eliminating arbitrary code from communications is to prevent communications from being treated as computations. Instead of creating a language from scratch, I have styled CSL after the Scala programming language [60]. I have implemented a simple source-to-source translator, which translates CSL specification code into pure Scala code for actual execution.

This chapter is organized as follows: Section 6.1 presents the abstract syntax of CSL. Design concerns and open questions are discussed in Section 6.2. CSL code examples for behaviors and communications are elaborated in Section 6.3. Section 6.4 describes the CSL translator. Section 6.5 summarizes this chapter.

6.1 Syntax

All valid Scala identifiers, types, variable declarations are valid in CSL; however, not all Scala expressions are valid. Particularly, I make only a small set of Scala expressions available in CSL. In addition, CSL defines its own statements, which can be used to specify complex types of communications. Scala expressions used in CSL can be found in most programming languages and take similar forms. One can use a different programming language to define their own CSL or define CSL from scratch without referring to other programming languages. In this sense, the language presented in this chapter is just for proof-of-concept purpose. I do not restrict programmers from developing their own CSL.

I first present basic Scala types used in CSL in Section 6.1.1. Then I introduce the Syntax of CSL from Section 6.1.2 to Section 6.1.5.

6.1.1 Scala Types

Basic Scala types used in CSL includes *Int*, *String*, *Any*, *ActorRef*, *List*, and *Map*. *Int* defines integers; *String* defines strings; *Any* is the super type of any type, which is similar to Java's superclass *Object*; *ActorRef*

defines a reference to a concurrent computation;[1] *List* defines a list of objects; and *Map* declares a pair of values in the form of (a, b), where a is the first value, b the second. *List* and *Map* can be used to build more complex types. For example, *List[Int]* defines a list of integers and *List[Any]* defines a list of object with the type of *Any*; *Map[Int, String]* defines a pair: the first value has the type of *Int* and the second has the type of *String*. CSL also uses Scala *tuple*, which can contain different types of elements. For example, `val pair = (100, "hello world")` defines a 2-tuple. To access the first element, we use `pair._1`, and the second, `pair._2`.

6.1.2 Main Body

The abstract syntax for specifying a communication is presented in Figure 6.1. In the syntax, I do not define the terms written in ***bold italic*** font because they should be self-explanatory. Keywords are written in `monospace` font.

A communication definition begins with the keyword `communication`. A communication can be one of two types: simple or composed. Either type of communication has a name, and is specified using a list of attributes, the logic of the communications – represented by lists of outlets and handlers for a simple communication, and represented by a list of existing communications for a composed communication – and finally an initialization method.

attributes declares a list of attributes, each of the form *attr_name: Scala_type*, where *attr_name* denotes an attribute name.

A simple communication has a list of outlets *outlet_list*. Each outlet has the type of `input` or `output` and has a behavior, which stipulates how an outlet processes incoming messages. The definition of a behavior is discussed shortly in Section 6.1.4. In addition, a communication may contain a list of handlers *handler_list*. Like outlets, each handler has a behavior. Generally, the behavior of a handler is more complex than the behavior of an outlet, because a handler may need to distinguish the sources of incoming messages, but an input outlet is not required to do so.

The initialization method *init* declares local variables, initializes these variables and sets up the initial rendezvous for a communication, which is automatically executed when the communication is launched. Statements used in this method are elaborated in Section 6.1.5.

A variable declaration takes two forms: `val` *variable_name (: Scala_type| = Init_Value)* defines a *constant* named *variable_name* with the type of *Scala_type* or the initial value *Init_Value*; `var` *variable_name (: Scala_type| = Init_Value)* defines a variable named *variable_name* with the type of *Scala_type* or the initial value *Init_Value*. The only difference between `val` declaration and `var` declaration is that a *constant* cannot be modified but a variable can. For example, both `val foo: Int` and `var bar: Int` define a variable; `foo` cannot be changed but `bar` can.

CSL also defines a keyword `this` used to refer to itself by a communication.

[1] In Scala, a concurrent computation is called *Actor*.

communication ::=

  `communication` ***name*** {

    `attributes:`{*attr_list*}

    comm

    initialization

  }

comm ::= *simple* | *composed*

simple ::= *outlet_list*

handler_list$^{(0|1)}$

composed ::= `comms:`{*comm_list*}

`composition:`{*composition_decl*}

attr_list ::= ***attr_name***: ***Scala_type***; *attr_list* | ***attr_name***: ***Scala_type***;

outlet_list ::= *outlet_decl*; *outlet_list* | *outlet_decl*;

outlet_decl ::= *outlet_type* `outlet:` ***outlet_name***(***behavior***)

outlet_type ::= `input` | `output`

handler_list ::= *handler_decl*; *handler_list* | *handler_decl*;

handler_decl ::= `handler:` ***handler_name(behavior***)

initialization ::=

  `init:` {

    var_list$^{(0|1)}$

    statement;*

  }

var_list ::= *var_decl*; *var_list* | *var_decl*;

var_decl ::= `val` ***variable_name*** (: ***Scala_type*** | = ***Init_Value***) |

`var` ***variable_name*** (: ***Scala_type*** | = ***Init_Value***)

Figure 6.1: Communication Specification Language: Main Body

6.1.3 Composition Definition

Figure 6.2 shows the syntax for composed communications. To define a composed communication, a list of communications *comm_list* must be provided, which defines a set of communications to be composed, and a composition glue *composition_decl* must be specified, which defines a set of outlets to be merged in the cases of *input-merge* and *output-merge* or a set of bindings in the case of *output-input merge*.

Each element in *comm_list* has the form of *comm_var: comm_type_name*. *comm_var* is a variable used in the definition of the composed communication; *comm_type_name* is a communication defined in CSL.

Composition glue for *output-input merge* is a list of bindings `bindings`, each of which is a 3-tuple. The three values in a tuple are output outlet, input outlet, and a behavior.

Composition glue for *input merge* and *output merge* has two parts: a list of input or output outlets `outlets` and a behavior `behavior` used in the newly created handler for the composition.

comm_list ::= ***comm_var***: ***comm_type_name***; *comm_list* |
comm_var: ***comm_type_name***;

composition_decl ::= `bindings: List(`*3_tuple_list*`)` |
`outlets: List(`*outlet_decl_list*`);`
`behavior: {`***behavior_var: behavior_type_name***`;}`

3_tuple_list ::= (***output_outlet, input_outlet, behavior***), *3_tuple_list* |
(***output_outlet, input_outlet, behavior***)

outlet_decl_list ::= ***outlet***, *outlet_decl_list* | ***outlet***

Figure 6.2: Communication Specification Language: Composition

6.1.4 Behavior Definition

Figure 6.3 shows the syntax for defining a behavior. A behavior definition starts with keyword `behavior` and follows a behavior name, a list of targets *targets*, a parameter list, and a definition body.

The parameter list *para_list* is a set of variables separated by comma. Variable declaration *var_decl* is defined in Figure 6.1.

The definition body has a method called `receive`, which has one parameter *msg* representing an incoming message and defines the process logic when a message *msg* arrives at the outlet or the handler having this behavior. *var_list* is defined in Figure 6.1 and *statement* is defined in Figure 6.4.

behavior ::= `behavior` ***behavior_name*** `(`***targets***, *para_list*`):` *behv_body*

para_list ::= *var_decl*, *para_list* | *var_decl*

behv_body ::= {
  var_list$^{(0|1)}$
  `receive(`***msg***`)` = {
    statement;*
  }
}

Figure 6.3: Communication Specification Language: Initialization Method and Behavior

6.1.5 Statements and Expressions

CSL does not allow arbitrary Scala code but only the statements and expressions defined in Figure 6.4 can be used in *init* and behavior definitions. I divide these statement into five groups.

Setter and Getter

- *communication*.`setAttr(`*attribute_map*) sets the attributes of *communication* with the provided attribute map with entries having the form of *(attribute_name, value)*.

- *communication*.`getInlets` gets the input outlets of *communication*.

Communication Statements

- `send(`*recipient*, *msg*`)` anonymously sends a message *msg* to a recipient named *recipient*

- `sendm(`*recipients*, *msg*`)` anonymously sends a message *msg* to multiple recipients named *recipients*

- `tell(`*sender*, *recipient*, *msg*`)` sends a message *msg* along with the sender's name *sender* to a recipient named *recipient* so that the recipient can reply to *sender*. Note that *sender* may be a third party other than the actual sender. Using `tell` when responses are expected.

- `tellm(`*sender*, *recipients*, *msg*`)` sends a message *msg* along with the sender's name *sender* to multiple recipients named *recipients* so that the recipients can reply *sender*. Similar to `tell`, *sender* may not be the actual sender. Using `tellm` when responses are expected.

statement ::=

communication.setAttr(***attribute_map***) |

communication.getInlets |

send(***recipient***, ***msg***) |

tell(***sender***, ***recipient***, ***msg***) |

sendm(***recipients***, ***msg***) |

tellm(***sender***, ***recipients***, ***msg***) |

multiSend(***recipient***, ***data_list***) |

multiTell(***sender***, ***recipients***, ***data_list***) |

createOutlet(*outlet_type,* ***behavior_name***) |

createHandler(***behavior_name***) |

change(***outlet_name, behavior_name***) |

inputs(***list_of_communications***, ***set_of_input_outlets***, ***behavior_name***) |

outputs(***list_of_communications***, ***set_of_output_outlets***, ***behavior_name***, ***targets***) |

outin(***list_of_communications***, ***bindings***) |

delay(***time***) |

subscribe(*trigger_type*, ***input_outlet_of_communication***) |

variable = *expression*

trigger_type ::= ***time_trigger*** | ***location_trigger*** | etc...

expression ::=

variable |

get(***Scala_list, nth***) |

set(***Scala_list, nth, value***) |

add(***element , Scala_list***) |

append(***element , Scala_list***) |

remove(***Scala_list, nth***) |

empty(***Scala_list***) |

isEmpty(***Scala_list***) |

size(***Scala_list***) |

null |

Scala_if_expression |

Scala_function_call_expression |

Scala_math_expression

Figure 6.4: Communication Specification Language: Statements and Expressions

- `multiSend(`*recipients*, *data_list*`)` anonymously sends the first element of *data_list* to the first recipient of *recipients*, the second element to the second recipient, and so on, until the last element to the last recipient.

- `multiTell(`*sender*, *recipients*, *data_list*`)` sends the first element of *data_list* to the first recipient of *recipients* along with the sender's name, the second element to the second recipient along with the sender's name, and so on, until the last element to the last recipient.

Outlet and Handler Operations

- `createOutlet(`*outlet_type, behavior_name*`)` creates an input or output outlet depending on the outlet type with the behavior named *behavior_name*.

- `createHandler(`*behavior_name*`)` creates a handler with the behavior named *behavior_name*.

- `change(`*outlet_name*, *behavior_name*`)` change the behavior of the outlet named *outlet_name* with the new behavior *behavior_name*.

In particular, I restrict outlets from using these operations, because although an outlet has a behavior, its role is primarily to serve as an interface between a communication and the processes using the communication. That is, an outlet cannot create an outlet, create a handler, or change the behavior of another outlet.

Composition Operations

- `inputs` merges a list of communication at a provided set of input outlets with the provided behavior.

- `outputs` merges a list of communication at a provided set of output outlets with the provided behavior.

- `outin` merges a list of communication with the provided bindings, each of which has the form of $(output\_outlet, input\_outlet, behavior)$.

Other Operations

- `delay(`*time*`)` delays the execution for defined *time* in milliseconds.

- `subscribe(`*trigger*, *input_outlet*`)` allows a communication to subscribe an external trigger which can send a message to the specified input outlet of the subscribing communication. Possible types of triggers can be time, location, signal strength, magnetic field, amount of light, amount of noise, etc.

- *variable* = *expression* assigns the value of *expression* to a variable named *variable*.

CSL expressions include *variable*, list operations, and four types of Scala expressions: `null`, `if` expressions, function call expressions, and math expressions.

List operations are functions for accessing or manipulating lists:

- `get(`*Scala_list, n*`)`: gets the n^{th} element of a list
- `set(`*Scala_list, n, value*`)`: sets the n^{th} element of a list to *value*.
- `add(`*Scala_list, elem*`)`: adds an element to the head of a list and returns a new list
- `append(`*Scala_list, elem*`)`: appends an element to the tail of a list and returns a new list
- `remove(`*Scala_list, n*`)`: removes the nth element of a list and returns the rest of list
- `empty(`*Scala_list*`)`: empties a list
- `isEmpty(`*Scala_list*`)`: checks whether a list is empty
- `size(`*Scala_list*`)`: determines the number of elements in a list

Scala lists are similar to arrays, which means all elements of a list have the same type.

6.2 Discussion

A number of design decisions were made in developing CSL. First, CSL is styled after Scala. Second, CSL tries to restrict arbitrary code in the definitions of behaviors and communications. Third, CSL is designed for clean and concise code.

Styled After Scala Currently, CSL is designed based on Scala. I developed a translator which translates CSL code into Scala code. The translator carries out textual substitutions and the generated code is compiled using Scala's compiler. Programmers can develop their own translator in order to translate CSL code into their preferred target languages and compile the generated code using compilers for the languages. To implement *interActors*, target languages must support message passing. In this thesis, Scala is chosen because Scala has native support for the Actor model through Akka actor library. Detailed discussions on choosing a target language can be found in Section 6.5.

Designing CSL from scratch is another option. In this scenario, CSL code would still need to be translated into a target language. In this sense, I do not see any advantage of designing CSL from scratch, whereas, there are a number of disadvantages. For example, we have to define CSL from the very basic elements, such as identifiers, literals, operators, and types, etc. Statements should be defined and a library for manipulating list should be implemented. These works can be avoided by having CSL styled after Scala.

Restrict Arbitrary Code In an effort to restrict the behavior of outlets and handlers to communication handling, loops are discouraged by the absence of support in CSL. More specifically, loops are avoided in the definitions of behaviors and communications so that separation of concerns is achieved. However, in some scenarios, repeated activities may be useful, which can be realized through triggers rather than functionally triggered loops, which are too powerful to be included in CSL. Triggers can send messages to

a communication which subscribes to them. Subscribing to a trigger makes a communication encapsulate the trigger in it. Figure 6.5 shows a time trigger encapsulated in a communication. Such a time trigger can periodically send a message to a subscribing communication potentially leading to periodic communication activity. In the prototype implementation, although I have implemented the `subscribe` method in `CSL` class, triggers for time, location, noise etc. have not yet been implemented.

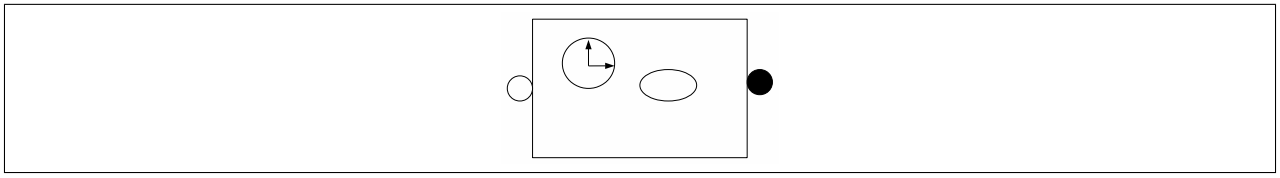

Figure 6.5: Trigger Example: Time

However, determined programmers could find indirect ways to include loops into a communication. An example of indirect loop can be found in Figure 4.11. For some scenarios that require functionally triggered loops, loops can be inserted into communications by programming them using Scala directly. The decision of disallowing loops in CSL may be imperfect. I would like to investigate the ramifactions of allowing loops in CSL in my future work.

Syntactic Sugar `sendm`, `tell`, `tellm`, `multiSend`, and `multiTell` are essentially syntactic sugar and can all be implemented using `send`. For example, `sendm` can be realized by repeatedly using `send` to send a message to a list of recipients; `tell` can be implemented using two `send`s to separately send a message and the sender's name; `multiSend` can be implemented by simultaneously iterating the recipient list and the data list and sending messages picked from the latter list to recipients chosen from the former list.

6.3 Examples

This section presents examples to show the expressiveness of CSL. Section 6.3.1 gives examples for defining behaviors. Section 6.3.2 describes a number of exemplary communications.

6.3.1 Behaviors

The examples presented in this section are *forwarder*, *filter*, *counter*, *applier*, *timer*, *router*, *sequencer*, *aggregator*, and *aggregatorOneOff*.

Forwarder

Figure 6.6 defines the behavior of a `forwarder` on the arrival of a message. A forwarder has `targets`. When receiving a message `msg`, a forwarder just simply sends the message to its targets, which is defined in `receive` method and realized through `sendm`.

```
1 behavior forwarder(targets): {
2   receive(msg) = {
3     sendm(targets, msg);
4   }
5 }
```

Figure 6.6: Behavior: Forwarder

Filter

Figure 6.7 defines *filter*, which has two parameters: `targets` and a function `cond`, which has the type of `Any => Boolean`. It is used to check whether an incoming message satisfies its defined condition. On the arrival of a message, it first is checked against `cond`. If it returns true, the message is sent to `targets`; otherwise, *filter* ignores the message.

```
1 behavior filter(targets, val cond: Any => Boolean): {
2   receive(msg) = {
3     if (cond(msg)) {
4       sendm(targets, msg);
5     }
6   }
7 }
```

Figure 6.7: Behavior: Filter

Counter

counter (Figure 6.8) counts the number of received messages. On receiving a message, a counter sends targets the number of received messages so far, which is the value of a local variable *i*.

```
1 behavior counter(targets): {
2   var i = 1;
3   receive(msg) = {
4     sendm(targets, i);
5     i = i + 1;
6   }
7 }
```

Figure 6.8: Behavior: Counter

Applier

Figure 6.9 defines *applier*. When receiving a message, it applies a programmer-provided function `f` having the type of `Any => Any` to the message and then sends the result to its targets.

Timer

Figure 6.10 defines *timer* which delays every received message for `time` milliseconds and then sends it to targets.

```
1 behavior applier(targets, val f: Any => Any): {
2   receive(msg) = {
3     sendm(targets, f(msg));
4   }
5 }
```

Figure 6.9: Behavior: Applier

```
1 behavior timer(targets, val time:Int): {
2   receive(msg) = {
3     delay(time);
4     sendm(targets, msg);
5   }
6 }
```

Figure 6.10: Behavior: Timer

Selector

selector is defined in Figure 6.11. It has a programmer-provided function `select`, which returns a list of recipients from `targets` based on a received message `msg`. Once a list of recipients is obtained, the received message is sent to those recipients. In this sense, different messages may be sent to different recipients, which is programmer-defined in function `select`. The concept of *selector* is similar to the concept of *routing* in Akka.

```
1 behavior selector(targets, val select: (Any, List[ActorRef]) => List[ActorRef]): {
2   receive(msg) = {
3     sendm(select(msg, targets), msg);
4   }
5 }
```

Figure 6.11: Behavior: Selector

Sequencer

Figure 6.12 defines *sequencer* which sends the first received message to the first participant in the list `parts`, waits for a response, sends the response to the second participant, and so forth until a response from the last participant is received and finally it is sent to a list of targets. After it is done, it discards any further incoming messages.

Aggregator

aggregator (Figure 6.13) accepts two functions: `cond` used to determine whether the aggregator should stop receiving messages, and `aggr` used to aggregate a result from all received messages so far. On arrival of a message, it is put into the aggregator's local message list `msgs`. Then the aggregator use `cond` to check the termination condition. If `cond` returns true, a result is aggregated from `msgs` and is sent to targets. The local message list `msgs` is emptied in order to be reused for next aggregation if any.

```
behavior sequencer(targets, val parts: List[ActorRef], val orders: List[Int]): {
  var next = 0;
  var done = false;

  receive(msg) = {
    if (done == false) {
       if (next == size(parts) - 1) {
          sendm(targets, msg);
          done = true;
       } else {
          send(get(parts, next), msg);
          next = next + 1;
       }
    }
  }
}
```

Figure 6.12: Behavior: Sequencer

```
behavior aggregator(targets, val cond: List[Any] => Boolean, val aggr: List[Any] => Any): {
  var msgs = List[Any]();
  receive(msg) = {
    msgs = append(msg, msgs);
    if (cond(msgs)) {
      sendm(targets, aggr(msgs));
      empty(msgs);
    }
  }
}
```

Figure 6.13: Behavior: Aggregator

AggregatorOneOff

aggregatorOneOff (Figure 6.14) is similar to *aggregator* except it only aggregates and sends a result once. This behavior is used in SOM2M.

```
behavior aggregatorOneOff(targets, val cond: List[Any] => Boolean, val aggr: List[Any] => Any): {
  var msgs = List[Any]();
  var sent = false;
  receive(msg) = {
    if (sent == false) {
       msgs = append(msg, msgs);
       if (cond(msgs)) {
         sendm(targets, aggr(msgs));
         sent = true;
       }
    }
  }
}
```

Figure 6.14: Behavior: One-Off Aggregator

6.3.2 Communications

This section presents CSL code for a number of communications presented in Chapter 5. Also, I present two examples for compositions: one for input merge and another for output merge. The example for output-input merge can be found in Chapter 7.

Broadcaster

A *broadcaster* enables a set of computations `participants` to send messages to another set of computations `recipients`. Figure 6.15 shows the CSL code.

`Broadcaster` has two attributes: `participants` are the senders and `recipients` are the message targets, one output outlet `out` targeting `recipients`, and one input outlet `in` targeting `out`. When such communication is launched, it notifies `participants` the name of the input outlet, so that they can use it to send messages. Upon receiving a message at its input outlet, the communication forwards it to its output outlet, which, in turn, forwards the message to `recipients`.

```
1  communication Broadcaster {
2    attributes: {
3      recipients: List[ActorRef];
4      participants: List[ActorRef];
5    }
6    output outlet: out(forwarder(recipients));
7    input outlet: in(forwarder(out));
8    init: {
9      sendm(participants, in);
10   }
11 }
```

Figure 6.15: Communication: Broadcaster

Router

A *router* communication described in Figure 6.16 selectively sends messages based on received messages. It defines two sets of recipients `recipients1` and `recipients2`. `participants` are senders and `select` is a provided function. It has two output outlets `out1` and `out2`, one handler `chooser`, and one input outlet `in`. The two output outlets have the behavior of `forwarder` and target `recipient1` and `recipient2`, respectively. The handler has the behavior of `selector`, which dynamically determines its target between `out1` and `out2` based on received messages. The select criteria is defined in function `select`, which chooses a list of recipients defined in `List[ActorRef]` based on a received message that has type of `Any`.

```
1  communication Router {
2    attributes: {
3      recipients1: List[ActorRef];
4      recipients2: List[ActorRef];
5      participants: List[ActorRef];
6      select: (Any, List[ActorRef]) => List[ActorRef];
7    }
8    output outlet: out1(forwarder(recipients1));
9    output outlet: out2(forwarder(recipients2));
10   handler: chooser(selector(List(out1, out2), select));
11   input outlet: in(forwarder(out1));
12   init: {
13     sendm(participants, in);
14   }
15 }
```

Figure 6.16: Communication: Router

Similar to a *broadcaster* communication, a *router* informs `participants` of the existence of the input outlet in its `init` method.

Multi-Origin Many-to-Many

The CSL version of MOM2M communication in Section 5.3 is defined in Figure 6.17. From the definition, a MOM2M has four attributes: `recipients`, which is the target of the output outlet, `participants` are message senders, and `cond` and `aggr` which are used by the aggregator behavior. The input outlet `in` targeting the handler `aggregator` which has the behavior of aggregator and targets the output outlet `out`.

```
communication Mom2m {
  attributes: {
    recipients: List[ActorRef];
    participants: List[ActorRef];

    cond: List[Any] => Boolean;
    aggr: List[Any] => Any;
  }
  output outlet: out(forwarder(recipients));
  handler: aggregator(aggregator(out, cond, aggr));
  input outlet: in(forwarder(aggregator));
  init: {
    sendm(participants, in);
  }
}
```

Figure 6.17: Communication: Multi-Origin Many-to-Many

Single-Origin Many-to-Many

The CSL version of Single-Origin Many-to-Many communication is defined in Figure 6.18. In the code, I assume behavior `spoutOneOff` has been defined.

```
communication Som2m {
  attributes: {
    recipients: List[ActorRef];
    participants: List[ActorRef];
    query: List[ActorRef];
    content: Any;

    cond: List[Any] => Boolean;
    aggr: List[Any] => Any;
  }
  output outlet: out(spoutOneOff(recipients, content));
  handler: aggregator(aggregatorOneOff(out, cond, aggr));
  input outlet: in(forwarder(aggregator));
  init: {
    tellm(in, participants, query);
  }
}
```

Figure 6.18: Communication: Single-Origin Many-to-Many

Composition – Input Merge

I present an example (Figure 6.19) which composes three broadcasters. The composed communication is `ins`. The three composing communications are `c1`, `c2`, and `c3`, and they are merged at their input outlets declared

as `outlets`. The handler merging them together has the behavior of *forwarder* and targets the merged outlets. Note that the handler can have a different behavior, such as *filter* and *timer*, etc. The composed communication has four attributes: `participants` are communicating parties which send messages through the composed communication, `recipients1`, `recipients2`, and `recipients3` are the recipients for the three composing communications, respectively. In the `init` method, the `participants` attribute of each of the three broadcasters is set to `null` because the participant is the handler created for the merge and will be available at the time of merging. The merge occurs at line 25. Line 27 notifies senders the input outlet of the composed communication so that they can use it to send messages.

```
1  communication ins {
2    comms: {
3      c1: Broadcaster;
4      c2: Broadcaster;
5      c3: Broadcaster;
6    }
7    composition: {
8      outlets: List(c1.in, c2.in, c3.in);
9      behavior: {
10        b:     forwarder(outlets);
11     }
12   }
13   attributes: {
14     participants: List[ActorRef];
15     recipients1: List[ActorRef];
16     recipients2: List[ActorRef];
17     recipients3: List[ActorRef];
18   }
19
20   init: {
21     c1.attrs(Map("recipients"->recipients1,"participants"->null));
22     c2.attrs(Map("recipients"->recipients2,"participants"->null));
23     c3.attrs(Map("recipients"->recipients3,"participants"->null));
24
25     Inputs(List(c1, c2, c3), outlets, b);
26
27     sendm(participants, this.getInlets);
28   }
29 }
```

Figure 6.19: Composition Example: Input Merge

Composition – Output Merge

The composed communication `outs` defined in Figure 6.20 composes three broadcasters at their output outlets. The three composing communications are `c1`, `c2`, and `c3`, and they are merged at their output outlets declared as `outlets`. The handler merging them together has the behavior of *forwarder*. Note that the handler can have a different behavior, such as *filter* and *timer*, etc. The composed communication has four attributes: `participants1`, `participants2`, and `participants3` are senders which send messages through the composed communication, `recipients` is the recipients of the composed communications. In particular, `participants1` use the input outlet of `c1` to send messages, `participants2` use the input outlet of `c2` to send messages, and `participants3` use the input outlet of `c3` to send messages,. In the `init` method, the `recipients` attribute of each of the three broadcasters is set to `null` because the recipient is the handler created for the merge and will be available at the time of merging. The merge occurs

at line 25. Lines 27-29 notify senders the input outlet of the composed communication so that they can use it to send messages.

```
1  communication outs {
2    comms: {
3      c1: Broadcaster;
4      c2: Broadcaster;
5      c3: Broadcaster;
6    }
7    composition: {
8      outlets: List(c1.out, c2.out, c3.out);
9      behavior: {
10        b:     forwarder(null);
11     }
12   }
13   attributes: {
14     recipients: List[ActorRef];
15     participants1: List[ActorRef];
16     participants2: List[ActorRef];
17     participants3: List[ActorRef];
18   }
19
20   init: {
21     c1.attrs(Map("recipients"->null,"participants"->participants1));
22     c2.attrs(Map("recipients"->null,"participants"->participants2));
23     c3.attrs(Map("recipients"->null,"participants"->participants3));
24
25     Outputs(List(c1, c2, c3), outlets, b, recipients);
26
27     sendm(participants1, c1.in);
28     sendm(participants2, c2.in);
29     sendm(participants3, c3.in);
30   }
31 }
```

Figure 6.20: Composition Example: Output Merge

6.4 Translator

I build a translator which translates CSL code into executable pure Scala code by carrying out textual substitution on CSL code. The translator consists of two separate applications: one is for translating behavior classes and the other is for translating communication classes. After translation, those classes can be used as Scala classes. In order to compile and execute the transferred code, the translated behavior classes must be put into `agents.envelope.behavior` package, and the translated communication classes must be put into `agents.envelope.communication` package.

Programmers can develop their own translator so as to convert CSL code to the code in the form of other programming languages so that they can use existing library of communications defined in CSL, such as *Broadcaster*, *Mom2m*, and *Som2m*, etc. Because of translators, an application developer can be dedicated to writing CSL code, which can be translated to various programming language code and thus can be used in different applications.

6.5 Summary

This chapter presents CSL, a specification language with the intention of restricting communications from including arbitrary code. Particularly, CSL does not allow loops in the definitions of communications and behaviors. Moreover, only statements and expressions defined CSL can be used. Through these two restrictions, CSL discourages arbitrarily complex code. The purpose of preventing arbitrary code is to prevent communications from being used to carry out computations. Otherwise, if communications can have arbitrary code, they will have the same capability as computations, which leads to mixing of concerns.

In this thesis, I use CSL to program *interActors*. As we have seen in Chapter 5, *interActors* can be programmed using Scala directly. In addition to Scala, *interActors* can be implemented using any language that supports message passing. It does not matter whether the message passing scheme is supported natively by the implementation language itself or through an external library and whether message passing is synchronous or asynchronous. For example, Microsoft has developed a library called Asynchronous Agents Library [57] for Visual C++. Computations can communicate with each other using this library. MPI is a message passing library and has implementations written in C, C++, and Fortran. Therefore, C and Fortran can be used to implement *interActors*. I expect that it should be possible to implement *interActors* using synchronous communication. This needs closer examination.

In the future, I would like to extend CSL with support of inheritance. That is, a communication can be defined as a subclass of another communication. Through subclassing, existing attributes, outlets, and handlers can be reused. Another enhancement to CSL could be a grammar checker, which checks whether CSL code violates the syntax of CSL. This would make it possible to identify grammar errors at the early stage of code development.

Chapter 7

Evaluation

This chapter evaluates our approach based on *interActors*. This work is a theoretical work and proposes a new programming model for separating processes' communication concerns from their functional concerns. Therefore, I evaluate *interActors* in terms of programmability, modularity, and reusability. Specifically, programmability is evaluated through comparison with Reo, a leading fundamental work on separation of interactions from computations. Modularity and reusability are evaluated through case studies, which also demonstrate programmability of *interActors*. Another purpose to do case studies is to show that *interActors* have potentials to be applied to a variety of application domains.

This chapter is organized as follows: Section 7.1 compares *interActors* with Reo, a leading interaction model by looking at solutions to particular communication problems using the two approaches. Section 7.2 presents three case studies to illustrate *interActors*' expressive power: gravitational n-body simulation, web services orchestration, and crowd-sourced services. Section 7.3 summarizes the discussions.

7.1 Comparison with Reo

This section compares *interActors* with Reo, a leading interaction model, which is a domain-specific language for programming and analyzing coordination protocols. Particularly, I compare *interActors* with an implementation of Reo described at [68]. I use the two examples presented in Chapter 1: Single-Origin Many-to-Many (SOM2M) and US presidential election. For each, I first show how a Reo connector is constructed and is used, and how an existing Reo connector can be reused. Then, I compare this with the construction and reuse of an *interActors* communication.

7.1.1 Problem: Single-Origin Many-to-Many

Here, I first build a Reo connector and use it to implement a Single-Origin Many-to-Many communication. Next, I develop a solution using interActors. I define a communication for this problem. To simplify this example, I assume that there are only three participants.

Reo solution

An implementation of the SOM2M communication using Reo is illustrated in Figure 7.1, in which we assume the use of a new channel type which we call *aggregator* which encapsulates a computational component (processes are called *components* in Reo literature). Although aggregation mechanisms can be achieved through embedding arbitrary computational components [12], the arbitrariness puts Reo on the slippery slope of mixing functional concerns with communication concerns. In the figure, labelled rectangles are the processes: the initiator and three participants p_1, p_2, and p_3, respectively. The dashed rectangle is the Reo connector, which is composed from five channels: *ab*, *cd*, *ef*, *gh*, and *mn*, respectively.[1] Each channel has two ends. Every channel end coincides with exactly one node, on which multiple channel ends may coincide. The circles are nodes. A node can be referred to by any channel end coincident on it. For example, either *b* or *e* can be used to refer to the node where the two ends meet. The behavior of the aggregator is to collect and aggregate the responses from participants and produce a boolean value determining whether the draft message should be sent. The gray circle in the connector broadcasts messages from the communication's initiator to participants.

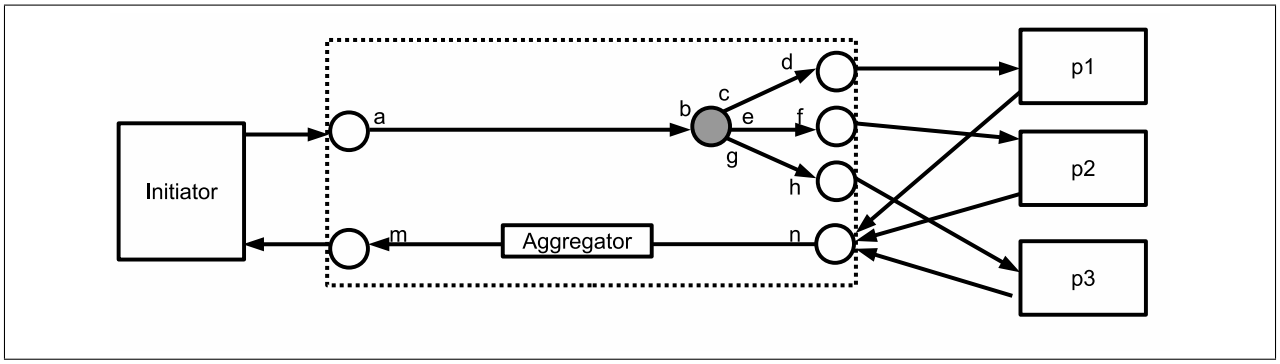

Figure 7.1: Reo Solution for SOM2M

The processes interact with each other through the connector, which implements the communication protocol between them. The initiator sends a request through node *a* to the three participants, which connect to the connector at node *d*, *f*, and *h*, respectively. After receiving a request, the three participants send their responses to node *n* that is also connected to the three participants. The aggregator channel collects responses and aggregates a boolean value and emits it through node *m*, which the initiator connects to. Upon receiving the boolean result, the initiator makes a decision on whether the draft message should be sent or discarded.

The pseudocode for building the connector is shown in Figure 7.2. The code first creates individual channels, and then builds the connector by joining channel ends together. To use this connector, each process must connect itself to it by calling the `connect` primitive. In order to connect to a connector, a process must know the name of the channel end.

[1]For more information on Reo channels and connectors, please refer to Section 2.3 in Chapter 2.

```
// create channels
create channel ab;
create channel cd;
create channel ef;
create channel gh;
create channel mn;
// construct the connector
join channel ends b and c;
join channel ends b and e;
join channel ends b and g;
```

Figure 7.2: Code for Building Reo SOM2M Connector

interActors solution

Using *interActors*, the initiator creates an instance of `Som2m` defined in Figure 7.3, initializes the communication by setting its attributes using `setAttr`, and launches it. The Scala code for doing this is shown in Figure 7.4. In the code, the list of recipients `recipients` and the list of participants `parts` are dynamic and are determined at run-time. Likewise, the query message `query`, the condition function `cond`, and the aggregation function `aggr` are provided by applications and thus are determined at run-time.

```
communication Som2m {
  attributes: {
    recipients: List[ActorRef];
    participants: List[ActorRef];
    query: List[ActorRef];
    content: Any;

    cond: List[Any] => Boolean;
    aggr: List[Any] => Any;
  }
  output outlet: out(spoutOneOff(recipients, content));
  handler: aggregator(aggregatorOneOff(out, cond, aggr));
  input outlet: in(forwarder(aggregator));
  init: {
    tellm(in, participants, query);
  }
}
```

Figure 7.3: Communication: Single-Origin Many-to-Many

```
val som2m = new Som2m(context)
som2m.setAttr(Map("recipients"->recipients, "participants"->parts,  "query"->query,
                  "content" -> draft, "cond" -> cond,   "aggr" -> aggr))
som2m.launch()
```

Figure 7.4: Code for Initiating and Launching SOM2M

Comparison

We compare *interActors* and Reo from the following perspectives:

Code Scalability The Reo solution is only for three participants. A larger number of participants would require a proportionately higher number of lines of code for creating and join channels. In other words, the number of lines of code for building the connector could be $O(n)$. In comparison, the number of participants

in the *interActors* solution is simply specified as a list at launch time, and thus the number of participants is determined at run time. Therefore, the number of lines of code for using a communication is $O(1)$.

Reusability First, the connector constructed in Reo solution cannot be easily reused because once constructed, Reo connectors offer static protocols. In other words, the number of channels and the number of nodes that can be connected by processes are fixed. In the connector in Figure 7.1, processes can connect to the connector at six nodes: a, d, f, h, m, and n. Among them, d, f, and h are used to notify participants. If, for example, a new participant is introduced, this connector cannot accommodate the new situation, and thus, a new connector must be constructed. On the contrary, `Som2m` can be easily reused by just simply resetting its attributes through `setAttr` method. Second, the *aggregator* channel in the Reo solution is very specific to this problem, with the termination policy (for instance) hard-coded. In other words, this channel cannot be reused for a problem that requires a different aggregation mechanism. In comparison, the communication in *interActors* solution is parametrized with aggregation functions, which can be specified at run-time, allowing for customization and reuse. For example, suppose the application uses *majority policy* first, and later on, the application requires to use *unanimous authorized policy*. For Reo, the existing connector does not meet the requirement and a new connector needs to be built; for *interActors*, programmers just pass a different condition function and a different aggregation function.

Modularity If the channels in the Reo solution are created and connected to the processes by one of the four processes (i.e., the initiator and the three participants), a mixing of concerns would occur. To avoid that, it would be better to have an external process create the channels. However, this would add more complexity to the system. Conversely, we encapsulate this concern into a single communication – `Som2m`, which is instantiated from a communication class that is from a communication library and the library can be easily extended by adding more communication classes.

7.1.2 Problem: US Presidential Election

Figure 7.5 illustrates the four levels in a US election. At each polling station, a subset of eligible voters vote. Each vote can be converted into a list with 0s for all but one of the candidates, who receives 1. The polling station operates from a start time to a finish time, eventually sending the list of sums of all the votes to the county level.

At the county level, vote totals are received from each of polling stations in the county. These totals are aggregated as they arrive, and once all stations have reported, the aggregated is reported to the state level. Partial results could be sent to media as well for a more gripping experience.

At the state level, similarly, county totals are received and aggregated; once all county totals have been received, the states electoral college votes are awarded to the winner, and the results are sent out to the national level. Again, partial totals could be sent to media.

Finally, at the national level, electoral college votes received from the states are aggregated until the total number of electoral college votes for one of the candidates reaches 270, at which time the result is announced.

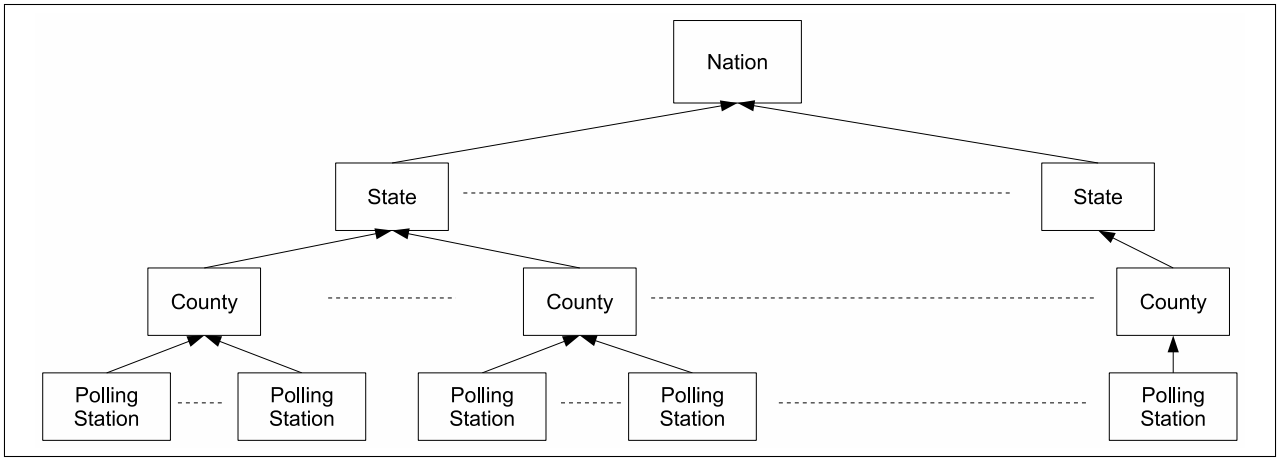

Figure 7.5: US Presidential Election

Reo Solution

To solve this problem using Reo, each voter is required to connect to a polling station that requires a channel to connect to the county it belongs to. Each county and state also requires a channel to connect to upper levels. We can build connectors for polling stations, counties, states, and the nation, respectively, and then connect these connectors to build a larger connector for the problem. Because polling stations, counties, states, and the nation may use different aggregation mechanism, at least, we need four types of Reo connectors. In the best-case scenario (Figure 7.6), these connectors can be reused. In other words, the connector created for polling stations can be used by all polling stations, the connector created for counties can be used by all counties, and so on. In the worst-case scenario, these connectors cannot be reused and we have to create a different type of connector for each polling station, county, state, and nation. This will lead to a large number of types of connectors.

```
1  create a connector type named pc for polling stations;
2  create a connector type named cc for counties;
3  create a connector type named sc for states;
4  create a connector type named nc for nation;
5
6  instantiate  a connector for polling station 1 from pc
7  ......
8  instantiate a connector for county 1 from cc
9  ......
10 instantiate a connector for state 1 from sc
11 ......
12 instantiate a connector for nation from nc
13
14 join polling station 1 to county 1
15 ......
16 join county 1 to state 1
17 ......
18 join state 1 to nation
19 ......
```

Figure 7.6: Code for Constructing a Connector for US Election

Furthermore, mechanisms such as making decisions when the result should be reported to the upper levels and how the results are aggregated can be implemented by embedding components (Reo's term for computa-

tions) in the connector. Again, although decision-making and aggregation mechanisms can be implemented by embedding components, Reo introduces arbitrariness to connectors and thus makes the boundary between communications and processes blur.

interActors Solution

The types of individual communications we need for implementing this election are special cases of MOM2M. This is the situation in which a number of parties want to send a collective message to some recipient(s). This type of a communication occurs naturally in a variety of scenarios, such as at mass celebrations, protests, etc. However, for it to happen, some type of a setup has to exist – such as a public square – where the individuals' messages can naturally aggregate into a group message. The communications required for nation, states, counties, and polling stations are slightly different from MOM2M, because each of them aggregates votes once and sends the aggregated results to its upper level once. For the national level, its upper level is to announce the election result to the public. Further, polling needs to end at some point in time, rather than wait until all messages have been received. Therefore, communications for polling stations are different from communications for the other three levels.

Figure 7.7 shows how this type of a communication can be programmed in CSL. `Mom2mOneOff` shows the communication code for the county, state, and national levels and `Mom2mTimer` shows the communication code for the polling station levels. The only difference between the two communication classes is the behaviors of their handlers. `Mom2mOneOff`'s handler has the behavior of `aggregatorOneOff` but `Mom2mTimer`'s handler has the behavior of `aggregatorTimer`.

`aggregatorTimer` is defined in Figure 7.8. The timed aggregator is similar to `aggregatorOneOff` defined in Figure 6.14. The only difference is that the timed aggregator checks the type of received messages. That is, once receiving a message, the timed aggregator checks if the message has the type of `Time`. If it does, the timed aggregator does not insert the message to its message list.

To construct an election communication requires composing the needed `Mom2mOneOff` and `Mom2mTimer` communications, appropriately parametrized with `cond` and `aggr` functions using provided attribute values. Figure 7.9 shows snippets of the code for composing polling station, county, state, and national communications to build an election communication. The communication for polling stations is created by instatiating `Mom2mTimer` communication type and the communication for counties, state, and nation is created by instantiating `Mom2mOneOff` communication type. The composing communications are declared, and then the required output-input merge bindings for the composition are specified. Attributes are defined. Next, the init function is provided, which first customizes each of the `Mom2mOneOff` and `Mom2mTimer` communications by setting its attributes – `cond` and `aggr` functions as well as `recipients` and `participants` lists as applicable – and then calls the `OutIn` composition primitive with the required `bindings`. Finally, the participants of the polling station level communications (i.e., the voters) are sent the names of the stations' input outlets to send their votes.

```
communication Mom2mOneOff {
  attributes: {
    recipients: List[ActorRef];
    participants: List[ActorRef];

    cond: List[Any] => Boolean;
    aggr: List[Any] => Any;
  }
  output outlet: out(forwarder(recipients));
  handler: aggregator(aggregatorOneOff(out, cond, aggr));
  input outlet: in(forwarder(aggregator));
  init: {
    sendm(participants, in);
  }
}

communication Mom2mTimer {
  attributes: {
    recipients: List[ActorRef];
    participants: List[ActorRef];

    cond: List[Any] => Boolean;
    aggr: List[Any] => Any;
  }
  output outlet: out(forwarder(recipients));
  handler: aggregator(aggregatorTimer(out, cond, aggr));
  input outlet: in(forwarder(aggregator));
  init: {
    subscribe(time_trigger);
    sendm(participants, in);
  }
}
```

Figure 7.7: Code: One-Off MOM2M and Timed MOM2M

Condition and aggregation functions for polling station, county, state, and nation are shown in Figure 7.10. `cond_p` defines the condition function for polling stations. For a particular station, it returns `true` when the station closes; otherwise, `false`. We assume that function `time()` returns the current time in milliseconds. `cond` function defines condition function for county and state. It returns `true` when the number of received aggregation messages equals to the required number `required`. `cond_n` function defines condition function for nation. It returns `true` as long as the number of electoral college votes for one candidate is greater than or equal to 270. `Ballot` class is used to keep track of the number of votes (electoral college votes at national level) for each candidate. Particularly, we use `seq` – an integer – to represent a candidate, and use `no` to record the number of votes for the corresponding candidate. `aggr_pc` defines the aggregation function for polling station and county. It calculates the number of votes received for each candidate and returns the aggregated result, which will be sent to the upper level. `aggr_st` defines the aggregation function for state. It returns the winner of a particular state with the number of electoral college votes for that state. `aggr_n` defines the aggregation function for nation. It computes the winner of the election.

A process can then use an election communication by instantiating it, setting its attributes, and finally launching it as shown in Figure 7.11. Each passed condition function and aggregation function are defined from an appropriate function defined in Figure 7.10. For example, `cond_st1` is defined as `cond(required, _)`, where `required` is the number of counties in that particular state; `aggr_st1` is defined as `aggr_st(nc, nev, _)`, where `nc` is the number of candidates and `nev` is the number of electoral college votes that the state has.

```
behavior aggregatorTimer(targets, val cond: List[Any] => Boolean,
                                  val aggr: List[Any] => Any): {
  var msgs: List[Any];
  var sent = false;
  receive(msg) = {
    if (sent == false) {
       if (msg.isInstanceOf[Time] == false) {
          msgs = append(msg, msgs);
       }
       if (cond(msgs) || msg.isInstanceOf[Time] == true) {
         sendm(targets, aggr(msgs));
         sent = true;
       }
    }
  }
}
```

Figure 7.8: Behaviors: Timed Aggregator

Comparison

Code Scalability and Reusability Now, compare this solution with the challenges in implementing such an interaction using Reo. Particularly, a large number of different types of *connector* would need to be created, which might be in proportion to the total number of polling stations, counties, states, and nation. In contrast, for *interActors*, we only need two types of communications, `Mom2mOneOff` and `Mom2mTimer`. Then we create communication objects for polling stations, counties, states, and nation by passing different parameters.

Modularity The aggregations required for counting the votes, and the decisions about when votes should be reported – which I argue belong on the communication side – would need to be placed inside the component as processes (called *components* in Reo literature). This unrestricted placing of arbitrary computational processes within a protocol, in our view, although necessary in Reo, shows that the model fails to adequately separate communication and computation concerns. In contrast, in our approach, there is a principled decision to treat limited types of message-handling related computational logic as a communication concern, and only code strictly meeting that criterion is allowed in outlet and handler behaviors.

7.1.3 Discussion

I used two examples, SOM2M and US election, to compare *interActors* and Reo. Particularly, I have compared them along three dimensions: code scalability, reusability, and modularity. To some extent, the comparisons depend on how well the Reo systems have been designed. I have tried my best effort to make a fair comparison between *interActors* and Reo by designing Reo solutions well. For each dimension, *interActors* has some advantages over Reo. However, for applications involving a small number of communication participants and when the number of participants is fixed, Reo may be a better choice, because it offers a graphical programming interface and there are a number of channels at programmers' disposal. One example of this would be an alternator [10] which imposes an ordering on the flow of data from multiple source processes to a destination process. The source processes alternately send a message to the destination process. When the

```
communication election {
 comms: { // communications to be composed
    national: Mom2mOneOff;
    state1: Mom2mOneOff; state2: Mom2mOneOff; ...
    county1: Mom2mOneOff; county2: Mom2mOneOff; ...
    station1: Mom2mTimer; station2: Mom2mTimer; ...
 }
 composition: { // output-input compositions
   bindings:List(
                ...
         (station1.out,county1.in, forwarder(county1.in)),
         (station2.out,county1.in, forwarder(county1.in)));
                ...
 }
 attributes: {
   recipients: List[ActorRef];
   participants1: List[ActorRef]; ...
   cond_nation: List[Any] => Boolean;
   cond_state1: List[Any] => Boolean; ...
   cond_county1: List[Any] => Boolean; ...
   cond_station1: List[Any] => Boolean; ...
   aggr_nation: List[Any] => Any;
   aggr_state1: List[Any] => Any; ...
   aggr_county1: List[Any] => Any; ...
   aggr_station1: List[Any] => Any; ...
   }

 init: {
   // customize Mom2mOneOff and Mom2mTimer communications
  national.attrs(Map("recipients"-> null, "participants"->null, "cond"-> cond_nation, "aggr"-> aggr_nation));

  state1.attrs(Map("recipients"-> null, "participants"->null, "cond"-> cond_state1, "aggr"-> aggr_state1));
    ...
  county1.attrs(Map("recipients"-> null, "participants"->null, "cond"-> cond_county1, "aggr"-> aggr_county1));
   ...
  station1.attrs(Map("recipients"-> null, "participants"->participants1,
                     "cond"-> cond_station1, "aggr"-> aggr_station1));
     ...

  // compose
  OutIn(List(national, state1, ... , county1,..., station1, ...), bindings);

  // notify participants
  sendm(participants1, station1.in);
  sendm(participants2, station2.in);
        ...
 }
}
```

Figure 7.9: Election Communication

number of source processes is smaller than 4, Reo provides a better solution. When the number of source processes is equal to or greater than 4, the number of lines of code for constructing an alternator grows fast making *interActors* a better choice. Furthermore, in Reo, programmers can construct connectors by drawing them. Reo also provides an animation tool which helps visualize how a particular connector works, leading to easy of understanding the function of the connector.

Reo protocols are fixed, which means that the number of participants must be known *a priori* and makes Reo a bad choice for reasoning about open systems. In contrast, *interActors* support dynamically evolving protocols. New outlets and handlers can be created on the fly to meet changing communication requirements. Additionally, *interActors* support conditional actions through behaviors. That is, a communication can be completed or an action can be triggered without requiring responses from all senders. To achieve the same effect using Reo, a process must be embedded into connectors. This arbitrariness puts Reo on the slippery

```
// cond function for Polling Station
def cond_p(expire:Time,l:List[Any]):Boolean= {
   if (time() >= expire) {
       true
   } else {
       false
   }
}
// cond function for county and state
def cond(required: Int, l: List[Any]): Boolean = {
      if (l.length == required) true
      else false
}
// cond function for nation
def cond_n(required: Int, l: List[Any]): Boolean = {
  var ballots = List.fill(nc)(0)
  l.asInstanceOf[List[Ballot]].foreach(x =>
       ballots=ballots.updated(x.seq,ballots(x.seq)+x.no))
  var terminate = false
  ballots.foreach(x => {
     if (x.no >= 270) {
        terminate = true
        break;
     }
  })
  terminate
}

class Ballot (s: Int, n: Int) {
  val seq = s
  val no = n
}
// aggr function for polling station and county
// nc: number of candidates
def aggr_pc(nc: Int, l: List[Any]): Any = {
   var ballots = List.fill(nc)(0)
   l.asInstanceOf[List[Ballot]].foreach(x =>
       ballots=ballots.updated(x.seq,ballots(x.seq)+x.no))
   ballots
}
// aggr function for state
// nc: number of candidates
// nev: number of electoral votes
def aggr_st(nc: Int, nev: Int, l: List[Any]): Any = {
    var ballots = List.fill(nc)(0)
    l.asInstanceOf[List[Ballot]].foreach(x =>
         ballots =  ballots.updated(x.seq, ballots(x.seq)+x.no)
    )
    var m = 0
    var seq = 0;
    ballots.foreach(x =>
         if (x.no > m) {
            m = x.no
            seq = x.seq
         }
    )
    val b = new Ballot(seq, nev)
    b
  }
// aggr function for nation
// nc: number of candidates
def aggr_n(nc: Int, l: List[Any]): Any = {
    var ballots = List.fill(nc)(0)
    l.asInstanceOf[List[Ballot]].foreach(x =>
         ballots =  ballots.updated(x.seq, ballots(x.seq)+x.no)
    )
    var m = 0
    var winner = 0;
    ballots.foreach(x =>
         if (x.no > m) {
            m = x.no
            winner = x.seq
         }
    )
    winner
  }
```

Figure 7.10: Cond and Aggr Functions

```
1 val e = new election();
2
3 e.setAttr(Map("recipients"->r,
4               "participants1"->parts1,"cond_nation"->cond_n, "aggr_nation"->aggr_n,
5               "cond_state1"->cond_st1, "cond_county1"->cond_c1, "cond_station1"->cond_s1,
6               "aggr_state1"->aggr_st1,  "aggr_county1"->aggr_c1, "aggr_station1"->aggr_s1,
7               ...));
8
9 e.launch();
```

Figure 7.11: Creating and Launching an Election Communication

slope of mixing functional concerns with communication concerns, blurring the boundary between processes and communications.

For execution, Reo protocols specified in a graphical language need to be compiled into a language, and scalability of the executable is a significant concern [43]. In *interActors*, communications can be implemented in CSL, which is essentially a restricted version of the Scala programming language. CSL code has to be translated into Scala code for execution. Communications can also be coded directly in Scala; however, CSL makes it both easier, by dispensing with some of the boilerplate, as well as restricts arbitrary computations from being included in a communication.

7.2 Case Studies

This section presents three case studies: gravitational n-body simulation, web services orchestration, and crowd-sourced services. Gravitational n-body simulation presented in Section 7.2.1 is solved using two approaches: using one simple communication and using one composed communication. Interestingly, the two approaches (or more precisely, the two communications) are identical to application programmers who use them. An example of web services orchestration is introduced in Section 7.2.2. I use a composed communication including four component communications to tackle the problem. All four component communications are instances of the same communication. The solution promotes modularity and reusability. I demonstrate that communications can be used as long-lived services in Section 7.2.3 through a crowd-sourced service for restaurant recommendation.

7.2.1 Gravitational n-Body Simulation

Suppose we have a large group of celestial objects, such as stars, planets, asteroids, and dust clouds in a galaxy, interacting with each other by exerting gravitational pull forces. The problem is to predict the position and velocity of every individual body so as to simulate the evolution of the galaxy. Each body has a mass and an initial position and velocity. Forces exerted on a body by other bodies cause it to move to a new position and to have a new velocity. Usually, the simulation includes a large number of iterations. In each iteration, we calculate the exerted forces by other bodies for every body, update their positions and velocities.

In a manager-workers implementation of this simulation [7], a number of worker agents are made responsible for computing and tracking the position and velocity of a block of bodies, so that the total number of bodies is roughly evenly divided between all agents. Because each worker agent exactly has one block of bodies, the number of blocks is the same as the number of workers. Suppose there are w workers, and thus the n bodies are divided into w blocks. Each worker carries out a task that is a pair and has the form (i, j) where $1 \leq i, j \leq w$. The task is to calculate the forces between the bodies in the two blocks. Tasks are generated by a manager agent. For each simulation step, the following process is executed:

1. The manager agent creates a list of tasks (each a pair of blocks of bodies).

2. Each worker obtains a task from the manager. Once a task is assigned to a worker, the manager removes the task from the list.

3. Workers, after receiving a task, calculate the forces between the bodies in the two blocks indicated by the task

4. By the time there is no task in the list, each worker has computed and locally collected forces for several pairs of blocks of bodies. Although all forces have been calculated, they are not known by all worker agents. Therefore, workers share those forces with others by sending their own copy of forces to all other workers.

5. Once receiving all forces calculated by all other workers, each worker aggregates these forces with its own copy of forces to get the total forces exerted on the bodies in its own block. Based on the aggregated forces, each worker computes new positions and velocities for its own block of bodies.

6. Once finishing computing new positions and velocities, each worker, again, shares new positions and velocities of its own block of bodies with all other workers.

7. Upon receiving updated positions of velocities of a block of bodies, each worker aggregates them with the positions and velocities of its local copy of bodies.

The above process repeats until the simulation steps have been done. In this implementation, to minimize the length of messages, each worker has a local copy of the positions, velocities, and masses of all bodies.

Figure 7.12 illustrates the above process. In the figure, we suppose there are three workers. Therefore, the bodies are divided into three blocks. The circles are agents as labelled. The lines between agents represent communications between them. The curve lines mean computations that agents execute. The manager creates a bag of tasks. Each task has the form of (i, j), where $1 \leq i, j \leq 3$. In other words, there are total six tasks: $(1, 1)$, $(1, 2)$, $(1, 3)$, $(2, 2)$, $(2, 3)$, $(3, 3)$.

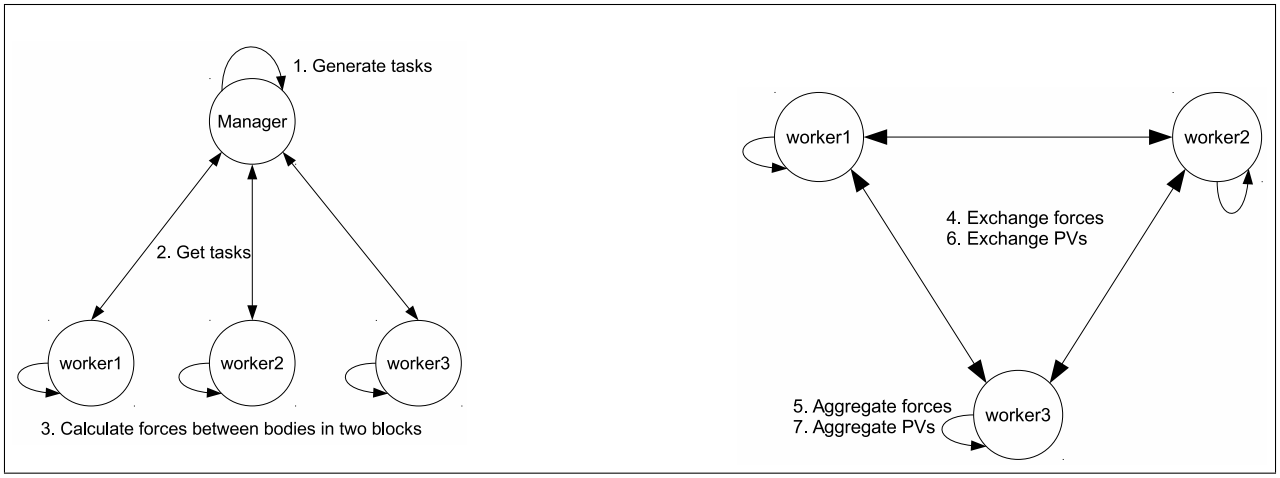

Figure 7.12: n-body Simulation – Manager-worker Implementation

Analysis

The worker's code in the manager-worker implementation is shown in Figure 7.13. In the code, we suppose there are n workers, w is the identity of the w^{th} worker, and the manager sends a $(0, 0)$ to indicate that the bag of tasks is empty. The number of simulation steps is $STEPS$.

By meticulously studying the manager-worker implementation, we find that aggregation tasks (i.e., force aggregation and position and velocity aggregation) should be treated as parts of communication, and thus, should be separated from workers. That is, the steps from 4 to 7 in the right side of Figure 7.12 are separated and treated as communications. Workers, consequently, only focus on calculating forces and positions and velocities. Our approach: *use a communication to carry out the aggregation tasks and exchange forces and positions and velocities*. Once finishing calculating forces or positions and velocities, each worker sends the result to the communication, which is responsible for aggregating and then sending the aggregated result to workers. Figure 7.14 illustrates this idea. In the figure, to make the figure simple and clean, I abstract away the outlets of the communication.

Now, I present *interActors* solutions. The problem is solved in two ways: using one simple communication and using one composed communication. In these solutions, we still use a manager to generate tasks. However, we transfer the aggregation tasks from the workers to a communication.

Solution 1: Using a Simple Communication

Figure 7.15 shows the solution using one simple communication, which has one input outlet, two handlers, and two output outlets. Once workers finish calculating forces or new PVs (Positions and Velocities), they send their local copies to the communication's input outlet. The input outlet executes the behavior `selector`, which selectively sends received messages to either the force handler if the received message is a force, or the PV handler if the received message is a PV. The force handler aggregates forces and the PV handler

```
1  task (int block1, int block2);
2
3  process Worker[w = 1 to n] {
4     for (s = 1 to STEPS) {
5        while (true) {
6          t = get a task from the manager;
7          if (t.block1 == 0) break;
8          calculate forces;
9        }
10       for (i = 1 to n st i != w) {
11          send local forces to ith worker;
12       }
13       for (i = 1 to n st i != w) {
14          receive forces from ith worker;
15          aggregate received forces to local forces;
16       }
17
18       calculate positions and velocities base on the aggregated forces;
19       for (i = 1 to n st i != w) {
20          send local bodies to ith worker;
21       }
22       for (i = 1 to n st i != w) {
23          receive bodies from ith worker;
24          move bodies;
25       }
26    }
27 }
```

Figure 7.13: Worker Code in Manager-worker Implementation

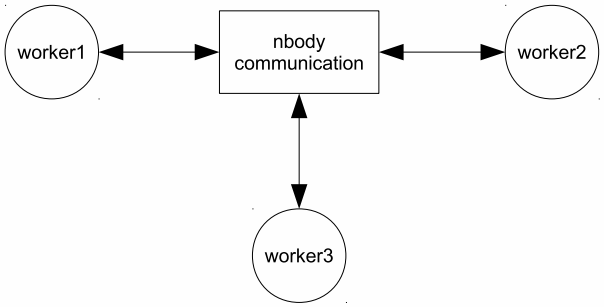

Figure 7.14: Solving n-body Simulation Using a Communication

aggregates positions and velocities of the bodies. After the aggregation is done, the handlers send the result to their corresponding output outlets, which, in turn, forward it to workers.

Figure 7.16 defines this communication named `nbodys` in CSL. `nbodys` has six attributes, namely, `workers` represents a list of workers; `condf` and `condpv` are used to determine whether the force handler should stop receiving forces and whether the PV handler should stop receiving positions and velocities, respectively; `aggrf` and `aggrpv` aggregate forces and PVs, respectively; `select` is used to select which handler a message is sent to. From the definition, we can see, the two output outlets have the behavior of *forwarder* and target the workers. The two handlers `hforce` and `hpv` handle force aggregation and PV aggregation, respectively, and target their corresponding output outlets. The input outlet targets the two handlers and has the behavior of *selector*, which selects one of handlers based on incoming messages as a target. When launched, the communication informs all workers the name of its input outlet.

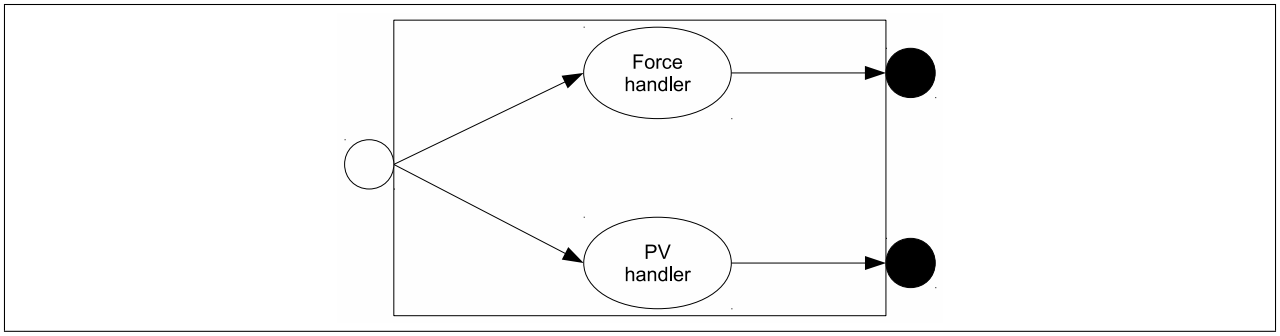

Figure 7.15: Solving n-body Simulation Using a Simple Communication

```
1  //Solution 1: use one simple communication
2  communication nbodys {
3    attributes: {
4      workers: List[ActorRef];
5      condf: List[Any] => Boolean;
6      condpv: List[Any] => Boolean;
7      aggrf: List[Any] => Any;
8      aggrpv: List[Any] => Any;
9      select: (Any, List[ActorRef]) => List[ActorRef];
10   }
11   output outlet: oforce(forwarder(workers));
12   output outlet: opv(forwarder(workers));
13   handler: hforce(aggregator(oforce, condf, aggrf)):
14   handler: hpv(aggregator(opv, condpv, aggrpv));
15   input outlet: in(selector(List(hforce, hpv), select));
16
17   init: {
18      sendm(workers, in);
19   }
20 }
```

Figure 7.16: n-body Simple Communication

To use this communication, we create an instance of it, set up its attributes, and launch it. Figure 7.17 shows the pseudocode for using it. In the code, we assume the variables assigned to attributes have already defined by application programmers.

```
1 nbs = new nbodys();
2 nbs.setAttr(Map("workers"->workers, "condf"->condf, "condpv"->condpv,
3                 "aggrf"->aggrf, "aggrpv"->aggrpv, "select"->select));
4 nbs.launch();
```

Figure 7.17: Using a Simple Communication for n-body Simulation

Worker Code Figure 7.18 shows pseudocode for workers. We suppose the workers have already know the communication's input outlet in the code. By comparing Figure 7.13 and Figure 7.18, we can see the code using a communication moves aggregation tasks to the communication and thus is much cleaner.

Solution 2: Using a Composed Communication

Figure 7.19 shows a composed communication used in this solution. The figure abstracts away outlets and handlers to only show the connections used to compose the communications. The composed communication has three sub-communications. *Router* is a communication with a handler executing the `selector` behavior,

```
task (int block1, int block2);
Communication c;
process Worker[w = 1 to n] {
   for (s = 1 to STEPS) {
      while (true) {
        t = get a task from the manager;
        if (t.block1 == 0) break;
        calculate forces;
      }
      send local forces to the c's input outlet;
      receive aggregated forces from the communication;
      calculate positions and velocities base on the aggregated forces;
      send local bodies to the c'input outlet;
      receive bodies from the communication;
      move bodies;
   }
}
```

Figure 7.18: Worker Code Using a Communication

which forwards received messages to one of the two *Mom2m* communications: one handles forces and another PVs.

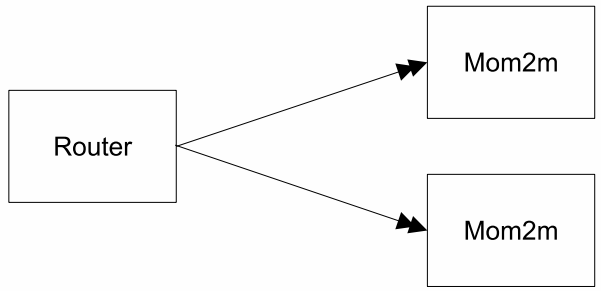

Figure 7.19: Solving n-body Problem Using a Composed Communication

Figure 7.20 shows the definition of the composed communication. In the code, we can see `nbodyc` is composed from three communications `rc`, `fc` and `pvc` by connecting `rc`'s `out1` to `fc`'s `in` and `rc`'s `out2` to `pvc`'s `in`. The worker's code is identical with the code for using one simple communication. The code for using this communication is shown in Figure 7.21 .

Interestingly, by comparing Figure 7.17 and Figure 7.21, we can see that the two pieces of code are almost identical except the names of the communication classes: one uses `nbodys` and the other uses `nbodyc`.

Discussion

Separation of Concerns In both *interActors* solutions, communication concerns are successfully separated from the functional processes, because the workers only need to compute forces and compute new positions and velocities without worrying about the aggregation, which, I argue, belongs to a communication. The consequences of separation are modularity and reusability which allows computations and communications to evolve independently. For example, for the workers, they may use different algorithms to compute forces; for the communication, they can employ different aggregation strategies.

```
//Solution 2: use a composed communication
communication nbodyc {
  comms: { // communications to be composed
    rc: Router;
    fc: Mom2m;
    pvc: Mom2m;
  }
  composition: { // composition glue
    bindings:List((rc.out1, fc.in, forwarder(fc.in)), (rc.out2, pvc.in, forwarder(pvc.in)));
  }
  attributes: {
    workers: List[ActorRef];
    condf: List[Any] => Boolean;
    condpv: List[Any] => Boolean;
    aggrf: List[Any] => Any;
    aggrpv: List[Any] => Any;
    select: (Any, List[ActorRef]) => List[ActorRef];
  }
  init: {
    // set attributes
    rc.attrs(Map("recipients"->workers,  "participants"->workers, "select"-> select));
    fc.attrs(Map("recipients"-> workers, "participants"->workers,  "cond"-> condf, "aggr"-> aggrf));
    pvc.attrs(Map("recipients"-> workers, "participants"->workers,  "cond"-> condpv, "aggr"-> aggrpv));
    // compose
    OutIn(List(rc, fc, pvc), bindings);
    // notify participants
    sendm(workers, rc.in);
  }
}
```

Figure 7.20: n-body Composed Communication

```
nbs = new nbodyc();
nbs.setAttr(Map("workers"->workers, "condf"->condf, "condpv"->condpv,
                "aggrf"->aggrf, "aggrpv"->aggrpv, "select"->select));
nbs.launch();
```

Figure 7.21: Using the Composed Communication for n-body Simulation

System Overhead The separation not only leads to modularity, but it also reduces the number of messages. For example, suppose there are n workers, the number of messages required for sharing forces is $n \times (n-1)$ in the manager-worker implementation because n workers need to send a message to the other $n-1$ workers. However, the number of messages is $2 \times n$ if they exchange forces through a communication: n messages sent from workers to the communication and n messages sent to the workers by the communication. The same analysis applies to the process of exchange positions and velocities.

However, when we look at the amount of network traffic, *interActors* introduce some network overhead. Suppose the calculated forces for each task is stored in an array and the length of force array is l, the i^{th} worker calculates b_i tasks and thus have b_i force arrays to share. In the manager-worker implementation, the amount of network traffic is: $\sum_{i=1}^{n} b_i \times l \times (n-1)$. Because the total number of tasks is $\frac{n \times (n+1)}{2}$, $\sum_{i=1}^{n} b_i = \frac{n \times (n+1)}{2}$. Therefore, the required network traffic is

$$\frac{n \times (n+1)}{2} \times (n-1) \times l \tag{7.1}$$

In the *interActors* solutions, the network traffic has two parts: from workers to the communication and from the communication to the workers. For the first part, the network traffic is $\sum_{i=1}^{n} b_i \times l = \frac{n \times (n+1)}{2} \times l$;

for the second part, because each message sent from the communication is $\frac{n \times (n+1)}{2} \times l$ and there are n messages, the network traffic is $\frac{n \times (n+1)}{2} \times l \times n$. Therefore, the total traffic is

$$\frac{n \times (n+1)}{2} \times (n+1) \times l \tag{7.2}$$

Comparing Equation 7.2 and Equation 7.1, we get the network overhead is:

$$n \times (n+1) \times l \tag{7.3}$$

Using the same analysis, the network overhead for exchanging positions and velocities is:

$$2 \times n \times l \tag{7.4}$$

where l is the length of position and velocity array for each block.

Complexity The introduction of a communication as a broker for sharing forces and positions and velocities adds a level of abstraction complexity to the system – the communication layer. One may argue that we can use a dedicated process to aggregate forces and positions and velocities. However, this solution, too, introduces a mediator. Moreover, the dedicated process not only aggregates forces but it also aggregates positions and velocities, which mixes concerns. Using two dedicated processes, one is for force aggregation and another is for position and velocity aggregation, resolves this problem; however, the workers need to know the two processes, whereas, in the solutions using a communication, the workers are only required to know one process – the input outlet of the communication. Worse, if an application requires more aggregation tasks, more dedicated processes are created. If using a communication, we still use one communication but add more handlers into it (for instance, in the case of using one simple communication).

7.2.2 Web Service Orchestration

To illustrate the concepts for orchestrating web services, I augment an example presented in [62]: consider a purchase system of a manufacturer. A product configuration of the manufacturer is made of a number of different types of parts. Each type of parts can be supplied by a number of suppliers. A buyer of the manufacturer works with a purchasing agent to fulfill the inventory requirements. The purchasing agent, in turn, contacts part suppliers. Once received offer responses from part suppliers, the agent reply to the buyer a proposal. Finally, the buyer determines whether making purchases or cancelling the request. Figure 7.22 shows this scenario. In the figure, each gray rectangle represents a set of part suppliers of the same kind. In this example, part suppliers and the buyer are considered as web services to be orchestrated, and the purchasing agent is the central composing process.

Analysis

Although this problem can be approached by allowing the agent to contact a list of part suppliers directly, this complicates the code for the agent because it should deal with suppliers for different parts. In this case,

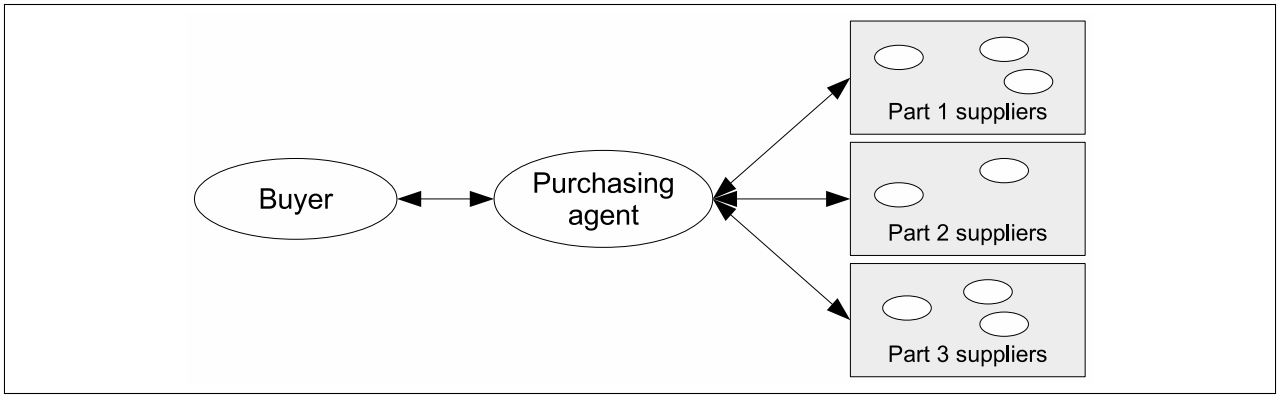

Figure 7.22: Part Purchasing Service

it should interact with three sets of part suppliers. Furthermore, if the product configuration changes, it is not easy to modify the code to accommodate new requirements because code for dealing with suppliers mixes together.

The purchasing agent can be treated as a communication, which is responsible for interacting with suppliers. To make the code of the purchasing agent simple and modular, instead of using one simple communication, we treat the purchasing agent as a composed communication. To distinguish different components in the composed communication, from here on, we call the purchasing agent the product purchasing agent and the composed communication purchasing communication. We introduce a part purchasing agent for each type of parts. The purchasing communication includes four purchasing agents: one product purchasing agent and three part purchasing agents. The goal of each part purchasing agent is to obtain the best offer from their part suppliers and then send the offer to the product purchasing agent. Once received offers from each part purchasing agent, the product purchasing agent can subsequently aggregates the best deal and proposes it to the buyer.

Interestingly, for each component in the purchasing communication, it carries out the following four similar tasks:

- receiving a request from a requester
- forwarding the request to participants
- gathering responses from participants
- aggregating a result and sending it to the requester

Therefore, the four purchasing agents are the same communication type, which is presented next.

Solution

In this section, I first define a communication called Two-way Multi-Origin Many-to-Many (MOM2M-2W). Then we use it to build the purchasing communication.

MOM2M-2W Communication An MOM2M-2W communication has two input outlets, two output outlets, and one handler as shown in Figure 7.23. An MOM2M-2W not only collects responses from participants like an MOM2M, it also sends requests to participants. The two input outlets `request_in` and `response_in` are used to receive requests from requesters and receive responses from participants, respectively; the two output outlets `request_out` and `response_out` are used to forward requests to participants and forward responses to requesters, respectively; the handler is an aggregator, which collects responses and aggregates a result.

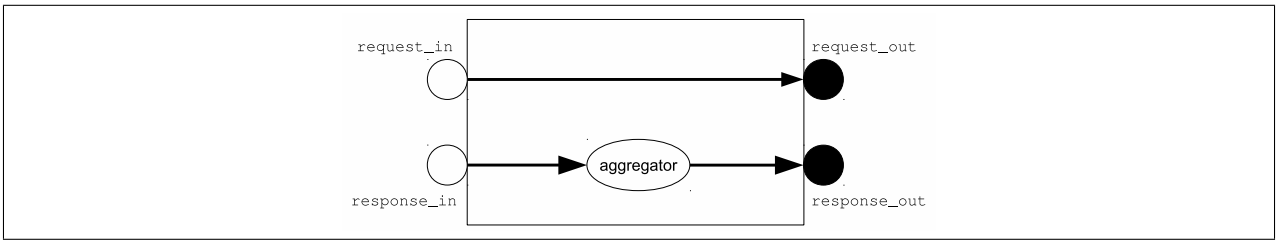

Figure 7.23: Two-way Multi-Origin Many-to-Many Communication

The code for the MOM2M-2W communication is shown in Figure 7.24. The communication accepts purchasing requests from `requesters` through the input outlet `request_in`, forwards the requests to `suppliers` through the output outlet `request_out`, gathers responses arrived at the input outlet `response_in`, and sends aggregated results to requesters through the output outlet `response_out`. The handler has the behavior of *aggregator*. `cond` function is used to determine when the communication should stop receiving responses and `aggr` function is used to aggregate a result from responses received so far. At the time of launching, the communication notifies the requesters the name of `request_in` and the suppliers the name of `response_in`.

```
1  communication Mom2m2w {
2    attributes: {
3      requesters: List[ActorRef];
4      suppliers: List[ActorRef];
5      cond: List[Any] => Boolean;
6      aggr: List[Any] => Any;
7    }
8    output outlet: request_out(forwarder(suppliers));
9    input outlet: request_in(forwarder(request_out));
10
11   output outlet: response_out(forwarder(requesters));
12   handler: aggregator(aggregator(response_out, cond, aggr));
13   input outlet: response_in(forwarder(aggregator));
14   init: {
15     sendm(requesters,request_in);
16     sendm(suppliers, response_in);
17   }
18 }
```

Figure 7.24: Code: Two-way Multi-Origin Many-to-Many Communication

Purchasing Communication Now we build the purchasing communication from `Mom2m2w`. The purchasing communication has four purchasing agents that are the type of `Mom2m2w`. Internally, the relationship among them is as follows: the product purchasing agent receives requests and forwards the requests to the

three part purchasing agents; once the part purchasing agents receive responses from suppliers and aggregate a result, they send it to the product purchasing agent. Externally, the product purchasing agent interacts with the buyer and the part purchasing agents communicate with part suppliers. Figure 7.25 shows the entire purchasing system. In the figure, PA represents the product purchasing agent; P1, P2, and P3 are the three part purchasing agents. The rectangle encapsulating PA, P1, P2, and P3 is the purchasing communication – the composed communication. To simplify the figure, I do not draw the outlets and the handler for each purchasing agent. Furthermore, I use double-arrows to represent the connections between communications. Recall double-arrows represent the connections (output-input merge) between communications. For example, the line with two double-arrows between PA and P1 means that there are two connections between them. That is, `request_out` of PA is connected to the `request_in` of P1 and `response_out` of P1 is connected to `response_out` of PA by referring to the definition shown in Figure 7.26.

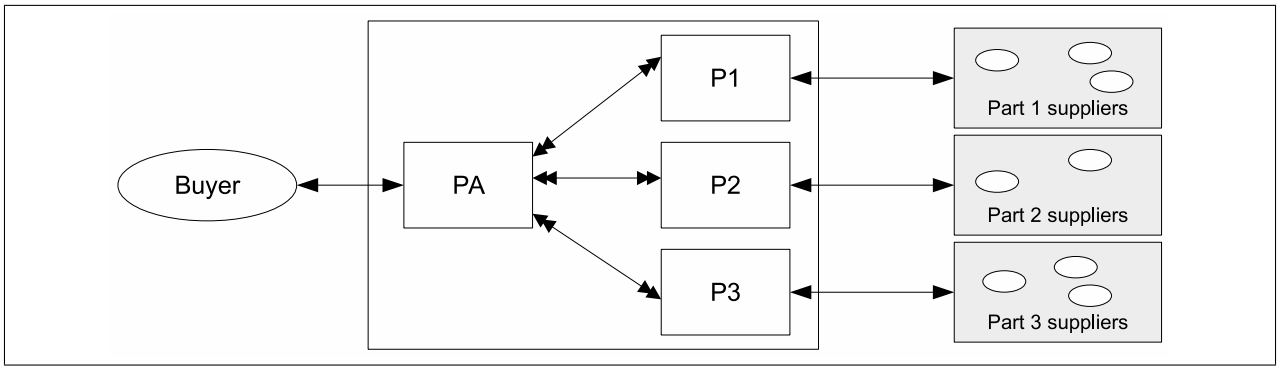

Figure 7.25: Manufacturer Purchasing System

Figure 7.26 defines the purchasing communication using CSL. The communication is named as `purchase` including four `Mom2m2w` communications: `pa`, `p1`, `p2`, and `p3`. `bindings` defines the composition glue between these communications. For example, line 10 defines the output outlet `request_out` of `pa` is connected to the input outlet `request_in` of `p1` through a handler with the behavior `forwarder` that targets the input outlet `request_in` of `p1`. `requesters` is a list of buyers that can send buy requests to the product purchasing agent `pa`. `suppliers1`, `suppliers2`, and `suppliers3` represents the three sets of part suppliers. For each agent, they have their own condition function and aggregation function: `conda` and `aggra` are for `pa`; `cond1` and `aggr1` are for `p1`, `cond2` and `aggr2` are for `p2`, and `cond3` and `aggr3` are for `p3`, which is reflected in line 35 - line 38. In these statements, some attributes are set to `null`, because they are not necessary to have an initial value since the communications will be composed in the later statement. Line 41 composes the four communications. Finally, the input outlets are sent to appropriate parties so that they can use them: the `request_in` of `pa` is sent to the requesters, `response_in` of `p1` is sent to `suppliers1`, `response_in` of `p2` is sent to `suppliers2`, and `response_in` of `p3` is sent to `suppliers3`.

```
communication purchase{
 comms: { // communications to be composed
   pa: Mom2m2w;
   p1: Mom2m2w;
   p2: Mom2m2w;
   p3: Mom2m2w;
 }
 composition: { // composition glue
   bindings:List(
                  (pa.request_out,p1.request_in, forwarder(p1.request_in)),
                  (pa.request_out,p2.request_in, forwarder(p2.request_in)),
                  (pa.request_out,p3.request_in, forwarder(p3.request_in)),
                  (p1.response_out,pa.response_in, forwarder(pa.response_in)),
                  (p2.response_out,pa.response_in, forwarder(pa.response_in)),
                  (p3.response_out,pa.response_in, forwarder(pa.response_in))
               );
 }
 attributes: {
   requesters: List[ActorRef];
   suppliers1: List[ActorRef];
   suppliers2: List[ActorRef];
   suppliers3: List[ActorRef];

   conda: List[Any] => Boolean;
   aggra: List[Any] => Any;
   cond1: List[Any] => Boolean;
   aggr1: List[Any] => Any;
   cond2: List[Any] => Boolean;
   aggr2: List[Any] => Any;
   cond3: List[Any] => Boolean;
   aggr3: List[Any] => Any;
 }
 init: {
   // set attributes
   pa.attrs(Map("requesters"-> requesters, "suppliers"->null,  "cond"-> conda, "aggr"-> aggra));
   p1.attrs(Map("requesters"-> null, "suppliers"->suppliers1,  "cond"-> cond1, "aggr"-> aggr1));
   p2.attrs(Map("requesters"-> null, "suppliers"->suppliers2,  "cond"-> cond2, "aggr"-> aggr2));
   p3.attrs(Map("requesters"-> null, "suppliers"->suppliers3,  "cond"-> cond3, "aggr"-> aggr3));

   // compose
   OutIn(List(pa, p1, p2, p3), bindings);
   // notify participants
   sendm(requesters, pa.request_in);
   sendm(suppliers1, p1.response_in);
   sendm(suppliers2, p2.response_in);
   sendm(suppliers3, p3.response_in);
 }
}
```

Figure 7.26: Code: Purchasing Communication

Discussion

Apparently, the communication-oriented solution leads to some overhead comparing to allowing the purchasing agent to interact with part suppliers directly. First of all, it introduces three part purchasing agents, which adds one more layer of complexity to the system and makes the system heavier. Second of all, the number of messages increases due to the newly introduced part agents, however, the number is constant, in this case is $2 \times 3 = 6$.

Whereas, I believe the benefits gained in this solution shadow its shortcomings. The benefits are a modular system and a reusable component – `Mom2m2w` communication. Because of the modularity, the system can be easily scaled up to handle more types of part suppliers.

Because of composition, the `Mom2m2w` communication can be used in different applications. For example, consider a client booking a vacation through a travel agent. The agent, in turn, contacts different service

providers in order to find a deal for her client. The vacation package should include a round-trip flight, a hotel room, and a car. The goal of the agent is to find the best deal for her client. In practice, in addition to prices, the client may have other constraints, such as departure and arrival time of a flight, location of a hotel, etc. This example is nearly the same as the manufacturer purchasing system and can be solved using the same solution as described above. The only difference is that this application may require different values to initiate attributes and functions.

7.2.3 Crowd-sourced Services

This section shows a communication can be used as a service. Consider a crowd-sensed restaurant recommendation service based on data collected from diners' smartphones. A long-lived service of this type would need to group devices based on their geographic locations, solicit data from their devices, aggregate data from those at the same restaurant, etc. Figure 7.27 illustrates this scenario. A client who wants to get reviews about restaurants in the system sends a request to the crowd-sourced service, which is a communication. The crowd-sourced service responds the client the reviews of all restaurants, and subsequently, the client can make her own decision based on the received reviews. The smartphone owners can send their reviews about a restaurant to the crowd-sourced service, which collects those reviews and calculates overall reviews for restaurants. In this example, we suppose the smartphone owners know where they send their reviews. For example, this information may be pre-set when a mobile application using the crowd-sourced service is installed on a smartphone.

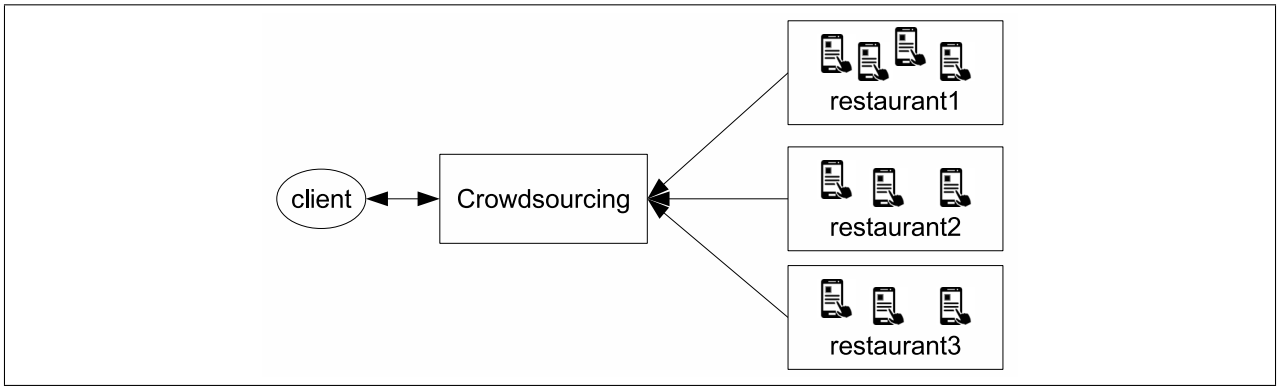

Figure 7.27: Restaurant-recommendation System

Figure 7.28 shows the crowd-sourced service communication that has two input outlets, one output outlet, and one handler. The two input outlets `request_in` and `response_in` are used by a client to send a request and by smartphones to send reviews, respectively. Both of them target the handler. The output outlet `response_out` is used to send reviews to the requester. The handler `recommender` not only accepts requests from clients and responds them with reviews of all restaurants, but it also collects reviews from smartphones and aggregates an overall review for each restaurant.

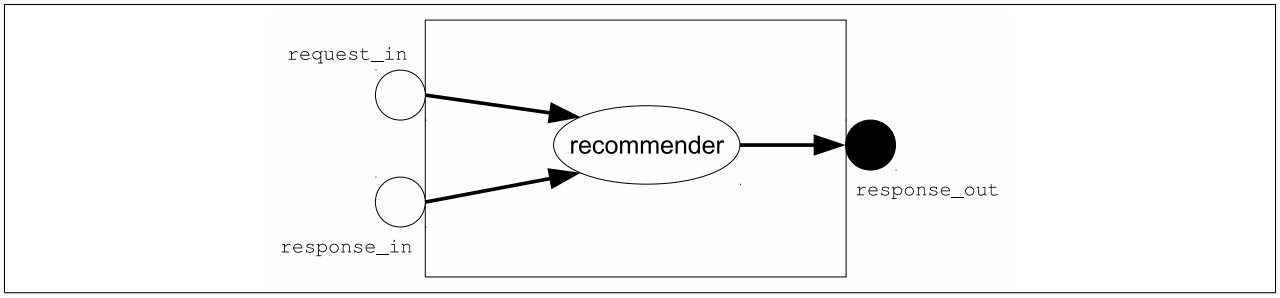

Figure 7.28: Crowd-sourced Service Communication

Figure 7.29 defines the crowd-sourced communication using CSL. Because we suppose the smartphones know the existence of the communication, we use a broker `broker` to hold the name of `response_in` so that any smartphone wants to send its reviews without knowing the name of `response_in` should contact the broker first to get it. We assume the name of the broker is a well-known name in the system. At the time of launching a crowd-sourced communication, the name of `response_in` is sent to the broker. In the definition, the output outlet `response_out` has the behavior of `connector` and the handler `hdr` has the behavior of `recommender`. We define these two behaviors next.

```
1 communication crowdsourcing {
2   attributes: {
3     broker: ActorRef;
4   }
5   output outlet: response_out(connector(null));
6   handler: hdr(recommender(response_out));
7   input outlet: request_in(forwarder(hdr));
8   input outlet: response_in(forwarder(hdr));
9
10  init: {
11    sendm(broker, response_in);
12  }
13 }
```

Figure 7.29: Code: Crowd-sourced Communication

recommender Behavior `recommender`, defined in Figure 7.30, does two things in accordance with different types of incoming messages:

1. When an incoming message is a name of an actor, it sends the reviews of all restaurants along with the name of the actor to its targets.

2. Otherwise, the incoming message is a tuple having two integers: the first one is the identity of a restaurant to be rated; the second one is the rate number. `recommender` computes a new overall reviews based on the incoming message.

`recommender` has two lists `nors` and `reviews`, which are used to hold the number of reviews for each restaurant and its current overall review, respectively. `recommender` calculates a new overall reviews for a restaurant from its current overall review, the incoming review, and the number of reviews received so

far (line 14). After obtaining the new overall review, `recommender` updates the two local lists `nors` and `reviews` using the new values.

```
1 behavior recommender(targets) {
2   var nors: List[Int]; // a list of the number of reviews for each restaurant
3   var reviews: List[Int]; // the overall reviews for each restaurant
4   receive(msg) = {
5     if (msg.isInstanceOf[ActorRef]) {
6       send(targets, (msg, reviews));
7     } else if (msg.isInstanceOf[(Int, Int)]) {
8      val ml = msg.asInstanceOf[(Int, Int)];
9      var id = ml._1;
10     var rate = ml._2;
11     var nrs = get(nors, id); // get the number of reviews
12     var review = get(reviews, id); // get the current review
13     // compute the new review
14     review = (review * nrs + rate) /(nrs + 1);
15
16    // set the new value for nors and reviews
17     set(nors, id, (nrs+1));
18     set(reviews, id, review);
19    }
20  }
21 }
```

Figure 7.30: Code: Recommender Behavior

connector The definition of behavior of `connector` is shown in Figure 7.31. An incoming message of `connector` has the type of `(ActorRef, List[Int])`. At the arrival of a message, `connector` retrieves the name of an actor and obtains a list of integers from the message. Then it sends the list of integers to the actor. In the crowd-sourced communication, `connector` sends out a list of reviews. Notice that `connector` ignores its passing targets `targets` but uses the actor obtained from incoming messages as a message target.

```
1 behavior connector(targets) {
2   receive(msg) = {
3     if (msg.isInstanceOf[(ActorRef, List[Int])]) {
4        val ml = msg.asInstanceOf[(ActorRef, List[Int])];
5        val t = ml._1;
6        val ranks = ml._2;
7        send(t, ranks);
8     }
9   }
10 }
```

Figure 7.31: Code: Connector Behavior

Discussion

This section uses the crowd-sourced service to show that communications can be used as long-lived services. I do not intend to use communications only to simulate complex social networking systems, such as Facebook, Twitter, and Linkedin. However, the communication part of those systems can be separated from computations and can be encapsulated in communications.

7.3 Summary

This chapter evaluated *interActors* through comparison and case studies. *interActors* make applications modular and makes applications' components independent and reusable. Section 7.1 compares *interActors* with Reo using two examples: SOM2M and US election. The comparison with Reo showed that *interActors* offer advantages in terms of programmability. We used communications to coordinate worker agents in an n-body simulation in Section 7.2.1, to orchestrate a number of web services in Section 7.2.2, and to provide a web service in Section 7.2.3. Among these five examples, three (i.e., US election, n-body simulation problem, and web services orchestration) used communication composition. From these examples, we can see communication composition not only facilitates reusability but also promotes modularity.

There are two levels of reusability: class level and object level. In the class level, a communication class can be reused to compose a more complex communication class. In the object level, a communication object can be reused by resetting its attributes. This chapter uses a number of examples to demonstrate reusability in the class level.

We see that *interActors* may also improve overall system performance by reducing the number of messages (Section 7.2.1). However, it may introduce overhead to some applications (Section 7.2.2). Because *interActors* add a layer onto the existing computation layer, for some applications, systems using *interActors* may generate more messages than equivalent systems which do not use *interActors*. For example, the simplest case is of two processes communicating with each other through a channel, where two messages are added to the system: one from the sender to the channel and the other from the channel to the receiver. A communication can be thought of as a dedicated coordinator. If we ignore messages within the communication, the increase in the number of messages is in proportion to the number of parties using the communication.

interActors provide a different way to program communication protocols. Especially, *interActors* enable building of libraries of communications which can be used in the future. *interActors* can be used in many types of applications. For example, as demonstrated in Section 7.2.3, a communication can be used as a service. Because communications can be composed using the three composition rules, *interActors* can be used for composing web services. Because *interActors* add an additional layer onto existing systems, it is inevitable that it has some system overhead. Therefore, applications which cannot tolerate this overhead for better programmability may not benefit from *interActors*.

To conclude, it is up to programmers to determine whether using *interActors* would be beneficial or not. Generally speaking, if programmers require programmability, modularity, and reusability, they should consider using *interActors* in their applications.

Chapter 8

Conclusion and Future Work

Communication is ubiquitous in concurrent systems, both for information exchange and for coordination. Without communication, concurrent computations would turn to standalone islands. In a variety of emerging applications, the interactions in concurrent computations are becoming more complex and varied, often requiring more complex process logic at run-time, such as aggregating and decision-making. However, code for handling those process logic is often mixed with the functional code in an application. Leaving such complex interactions to be managed by the communicating computations complicates code, and hampers reusability and modularity. Although significant advances have been made in separating communication concerns of computations from their functional concerns, existing approaches create static protocols which cannot evolve over the course of interactions.

In this thesis, I developed a way for separating communications of computations from their functional concerns. Communications are treated as *first-class* objects and consist of two types of active objects: outlets and handlers. There are two types of outlets: *input* and *output*. Processes can send messages to a communication through the input outlets of the communication that they connect to, and can receive messages through the output outlets. Handlers carry out communication logics, which are not visible to external observers. Because outlets and handlers are active and essentially drive the communications, we say that communications are *self-driven*. A primitive communication is just a channel to connect two processes.

Complex communications can be built by composing simpler communications using three composition rules: *input merge*, *output merge*, and *output-input merge*. Making communications composable offers a number of benefits. The main benefit of composing communications is to facilitate the reuse of designs and implementations. Further, composed communications can be reasoned by reasoning their simpler composing communications, offering potentials to study the entire system easier. Last but not least, constructing complex communications by composing simpler communications enables developers to create libraries of novel types of communications, hence, developers can build a complex communication using simpler communications already existed in libraries instead of implementing it from scratch.

Operational semantics were developed by extending the actor model with support for complex communications. I applied the concept of computational reflection. A layer called the communication layer is added onto the existing actor layer. The communication layer is where complex communications occur and the

actor layer supports actors. Compositional semantics are presented which formalize the three composition rules.

I prototyped *interActors* using Scala and the Akka actor library. A number of example communications were given to illustrate *interActors*. The programming style using communications can be called Communication-Oriented Programming (COP).

I developed a special language called Communication Specification Language (CSL) with the intention of restricting arbitrarily complex code in communications. Specifically, CSL does not allow loops and thus guarantees that a communication terminates within a finite number of execution steps. I also developed a translator to translate CSL code into executable Scala code.

interActors are evaluated using case studies and by comparison with Reo, a leading coordination model, to demonstrate ease of programmability, reusability, and modularity.

8.1 Implications

There are a variety of applications which have important implications for *interActors*. Here, I highlight Massive Open Online Courses (MOOCs), internet of things, and crowd-sourced services.

MOOCs have been steadily gaining popularity. In a MOOC, interactions between students and teachers can be complex and challenging. For example, because there are a large number of students, it is impossible for a MOOC teacher to answer each student's question. One solution to this dilemma is that teachers only answer the questions that most students care about. Those questions could be selected through voting. Another challenge is that when there is a question available, when teachers should be notified? If a teacher frequently stops her lecture to answer questions, the delivery of the lecture would be disruptive and the student experience would be miserable.

Internet of Things (IoT) [45] involves a wide range of everyday objects that have embedded computing devices, and those objects generate, collect, and exchange data. Decision-making and aggregation mechanisms are often required in an IoT system [15], and considering those mechanisms as communications concerns would lead to a better solution.

There are many crowd-sourced services, such as change.org [20], Be My Eyes [30], and Airbnb [5], etc. These services collect data from their users, organize data, and aggregate data, which can be handled separately from the functional concerns of processes. *interActors* offer a way of developing crowd-sourced services using communications, which could be a more elegant way.

8.2 Future Directions

There are three important future directions which I would like to pursue. First, formally studying of the properties of systems based on *interActors*, second, make enhancements and improvements in CSL, and third, an exploration of interesting application domains.

I would like to investigate the properties of communications. CSL can be extended with support for inheritance. I would like to build a grammar checker for CSL so that grammar errors can be found in the early stage but not at the compile time when compiling translated code in other programming languages. Finally, the effectiveness of disallowing loops in CSL code in discouraging inclusion of functional concerns in communications will be studied, along with other opportunities for appropriately restricting CSL.

Among application domains mentioned in Section 8.1, I would like to explore opportunities to offer communications as services, and price them based on the resource demands they place on the system. For example, cloud services companies can build their services based on *interActors* and precisely reason about the actual cost of providing certain types of communications.

Other possible domains I am interested in include web services, big data, and wireless sensor networks. Communications can be used to compose web services as demonstrated in Chapter 7. A composed web service can be further composed with other web services using communications, to create composite web services as described in [47]. In big data and wireless sensor networks, collective communication patterns [21] are often involved, which can be easily encapsulated in communications.

References

[1] João Abreu and José Luiz Fiadeiro. A Coordination Model for Service-Oriented Interactions. In *Proceedings of the 10th International Conference on Coordination Models and Languages*, pages 1–16. Springer-Verlag, 2008.

[2] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.

[3] Gul Agha and Christian Callsen. Actorspace: Open Distributed Programming Paradigm. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 23–32, New York, NY, USA, 1993. ACM.

[4] Gul Agha, Ian Mason, Scott Smith, and Carolyn Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 7(1):1–72, January 1997.

[5] Airbnb. Airbnb. `http://www.airbnb.com/`.

[6] Abdaladhem Albreshne, Patrik Fuhrer, and Jacques Pasquier. Web Services Orchestration and Composition, 2009.

[7] Gregory Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

[8] Tony et.al. Andrews. Business Process Execution Language for Web Services. 2003.

[9] Farhad Arbab. The IWIM Model for Coordination of Concurrent Activities. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 34–56. Springer Berlin Heidelberg, 1996.

[10] Farhad Arbab. Reo: A Channel-based Coordination Model for Component Composition. *Mathematical Structures in Computer Science*, 14(3):329–366, June 2004.

[11] Farhad Arbab. Puff, The Magic Protocol. In Gul Agha, Olivier Danvy, and Jos Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems*, volume 7000 of *Lecture Notes in Computer Science*, pages 169–206. Springer Berlin Heidelberg, 2011.

[12] Farhad Arbab, Lăcrămioara Aştefănoaei, Frank S. Boer, Mehdi Dastani, John-Jules Meyer, and Nick Tinnermeier. Reo Connectors as Coordination Artifacts in 2APL Systems. In TheDuy Bui, TuongVinh Ho, and QuangThuy Ha, editors, *Intelligent Agents and Multi-Agent Systems*, volume 5357 of *Lecture Notes in Computer Science*, pages 42–53. Springer Berlin Heidelberg, 2008.

[13] Mark Astley, Daniel C. Sturman, and Gul Agha. Customizable Middleware for Modular Distributed Software. *Communication of ACM*, 44(5):99–107, 2001.

[14] Francoise Baude, Denis Caromel, Ludovic Henrio, and Matthieu Morel. Collective Interfaces for Distributed Components. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 599–610. IEEE Computer Society, 2007.

[15] Chiara Bodei, Pierpaolo Degano, Gian-Luigi Ferrari, and Letterio Galletta. Where Do Your IoT Ingredients Come From? In *International Conference on Coordination Languages and Models*, pages 35–50. Springer, 2016.

[16] Boomerang. Schedule Email to be Sent Later in Gmail. http://www.boomeranggmail.com/l/schedule-an-email.html. Accessed: 2016-10-17.

[17] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The Fractal Component Model and Its Support in Java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.

[18] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Reactive Tuple Spaces for Mobile Agent Coordination. In Kurt Rothermel and Fritz Hohl, editors, *Mobile Agents*, volume 1477 of *LNCS*, pages 237–248. Springer, 1998.

[19] Christian Callsen and Gul Agha. Open Heterogeneous Computing in ActorSpace. *Journal of Parallel and Distributed Computing*, 21(3):289–300, 1994.

[20] change.org. change.org. http://www.change.org/.

[21] Nicholas Chen, Rajesh Kumar Karmani, Amin Shali, Bor-Yiing Su, and Ralph Johnson. Collective Communication Patterns. In *Workshop on Parallel Programming Patterns (ParaPLOP)*, 2009.

[22] Gianpaolo Cugola and Alessandro Margara. *High-Performance Location-Aware Publish-Subscribe on GPUs*, pages 312–331. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[23] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic Choreographies. In *International Conference on Coordination Languages and Models*, pages 67–82. Springer, 2015.

[24] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communication of ACM*, 51(1):107–113.

[25] Enrico Denti, Antonio Natali, and Andrea Omicini. On the Expressive Power of a Language for Programming Coordination Media. In *Proceedings of the 1998 ACM Symposium on Applied Computing*, SAC '98, pages 169–177, New York, NY, USA, 1998. ACM.

[26] Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Dimitris Mostrous, and Nobuko Yoshida. Objects and Session Types. *Information and Computation*, 207(5):595 – 641, 2009.

[27] Peter Dinges and Gul Agha. Scoped Synchronization Constraints for Large Scale Actor Systems. In *Proceedings of the 14th international conference on Coordination Models and Languages*, COORDINATION'12, pages 89–103, Berlin, Heidelberg, 2012. Springer-Verlag.

[28] Aiden Dipple, Kerry Raymond, and Michael Docherty. Stigmergy in Web 2.0: a Model for Site Dynamics. In *Proceedings of the 3rd Annual ACM Web Science Conference*, WebScience 2012, pages 86–94, 2012.

[29] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131.

[30] Be My Eyes. Be my eyes. http://www.bemyeyes.org/.

[31] Svend Frølund. *Coordinating Distributed Objects : An Actor-based Approach to Synchronization.* MIT Press, 1996.

[32] Svend Frølund and Gul Agha. Abstracting Interactions Based on Message Sets. In *Object-Based Models and Languages for Concurrent Systems*, LNCS, pages 107–124. Springer-Verlag, 1995.

[33] David Gelernter and Nicholas Carriero. Coordination Languages and Their Significance. *Communication of ACM*, 35(2):96–107, February 1992.

[34] Hongxing Geng and Nadeem Jamali. Supporting Many-to-Many Communication. AGERE! '13, pages 81–86. ACM, 2013.

[35] Google. Priority Inbox. `https://support.google.com/mail/answer/186531`. Accessed: 2016-10-17.

[36] Sebastian Gutierrez-Nolasco and Nalini Venkatasubramanian. A Reflective Middleware Framework for Communication in Dynamic Environments. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems: CoopIS, DOA, and ODBASE*, pages 791–808. Springer, 2002.

[37] Sebastian Gutierrez-Nolasco and Nalini Venkatasubramanian. A Reflective Middleware Framework for Communication in Dynamic Environments. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*, volume 2519 of *Lecture Notes in Computer Science*, pages 791–808. Springer Berlin Heidelberg, 2002.

[38] C. A. R. Hoare. Communicating Sequential Processes. *Communication of ACM*, 21(8):666–677, August 1978.

[39] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 273–284. ACM, 2008.

[40] Yongqiang Huang and Hector Garcia-Molina. Publish/Subscribe in a Mobile Environment. *Wireless Networks*, 10(6):643–652.

[41] Shams M. Imam and Vivek Sarkar. Selectors: Actors with Multiple Guarded Mailboxes. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents, and Decentralized Control*, AGERE! '14, pages 1–14, New York, NY, USA, 2014. ACM.

[42] Nadeem Jamali and Hongxing Geng. A Mailbox Ownership Based Mechanism for Curbing Spam. *Computer Communications*, 31(15):3586 – 3593, 2008.

[43] Sung-Shik T. Q. Jongmans, Sean Halle, and Farhad Arbab. Reo: A Dataflow Inspired Language for Multicore. In *2013 Data-Flow Execution Models for Extreme Scale Computing*, pages 42–50, Sept 2013.

[44] Stephen Kell. Rethinking Software Connectors. In *International Workshop on Synthesis and Analysis of Component Connectors: In Conjunction with the 6th ESEC/FSE Joint Meeting*, SYANCO '07, pages 1–12. ACM, 2007.

[45] Hermann Kopetz. *Internet of Things*, pages 307–323. Springer US, Boston, MA, 2011.

[46] Kung-Kiu Lau, Mario Ornaghi, and Zheng Wang. A Software Component Model and Its Preliminary Formalisation. In *International Symposium on Formal Methods for Components and Objects*, pages 1–21. Springer, 2005.

[47] Kung-Kiu Lau and Cuong Tran. Composite Web Services. In *Emerging Web Services Technology, Volume II*, pages 77–95. Springer, 2008.

[48] Kung-Kiu Lau, Perla Velasco Elizondo, and Zheng Wang. Exogenous Connectors for Software Components. In *Proceedings of the 8th International Conference on Component-Based Software Engineering*, CBSE'05, pages 90–106, Berlin, Heidelberg, 2005. Springer-Verlag.

[49] Vitaliy Liptchinsky, Roman Khazankin, Hong-Linh Truong, and Schahram Dustdar. Statelets: Coordination of Social Collaboration Processes. In *Proceedings of the 14th International Conference on Coordination Models and Languages*, pages 1–16. Springer-Verlag, 2012.

[50] Andoni Lombide Carreton and Theo D'Hondt. A Hybrid Visual Dataflow Language for Coordination in Mobile Ad-hoc Networks. In *Proceedings of the 12th International Conference on Coordination Models and Languages*, pages 76–91. Springer-Verlag, 2010.

[51] Pattie Maes. Computational Reflection. In Katharina Morik, editor, *GWAI-87 11th German Workshop on Artifical Intelligence*, volume 152 of *Informatik-Fachberichte*, pages 251–265. Springer Berlin Heidelberg, 1987.

[52] Francisco Maia, Miguel Matos, José Pereira, and Rui Oliveira. Worldwide Consensus. In *Proceedings of the 11th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, pages 257–269. Springer-Verlag, 2011.

[53] Thomas W. Malone and Kevin Crowston. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, 26(1):87–119, March 1994.

[54] Marco Mamei and Franco Zambonelli. Programming Stigmergic Coordination with the TOTA Middleware. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '05, pages 415–422, New York, NY, USA, 2005. ACM.

[55] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a Taxonomy of Software Connectors. In *Proceedings of the 22Nd International Conference on Software Engineering*, pages 178–187. ACM, 2000.

[56] Chris Metz. IP Anycast: Point-to-(any) Point Communication. *IEEE Internet Computing*, 6(2):94–98, March 2002.

[57] Microsoft. Asynchronous Agents Library. `https://msdn.microsoft.com/en-us/library/dd492627.aspx`.

[58] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[59] Robin Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[60] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language. Technical report, 2004.

[61] Andrea Omicini and Enrico Denti. From Tuple Spaces to Tuple Centres. *Science of Computer Programming*, 41(3):277 – 294, 2001.

[62] Chris Peltz. Web Services Orchestration and Choreography. *Computer*, 36(10):46–52, 2003.

[63] Eline Philips, Jorge Vallejos, Ragnhild Van Der Straeten, and Viviane Jonckers. Group Orchestration in a Mobile Environment. In *Proceedings of the 14th International Conference on Coordination Models and Languages*, pages 181–195. Springer-Verlag, 2012.

[64] Eline Philips, Ragnhild Van Der Straeten, and Viviane Jonckers. Now: a Workflow Language for Orchestration in Nomadic Networks. In *Proceedings of the 12th International Conference on Coordination Models and Languages*, pages 31–45. Springer-Verlag, 2010.

[65] Esmond Pitt and Kathy McNiff. *Java RMI: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[66] Aleksandar Prokopec and Martin Odersky. Isolates, Channels, and Event Streams for Composable Distributed Programming. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 171–182. ACM, 2015.

[67] Shangping Ren, Yue Yu, Nianen Chen, Kevin Marth, Pierre-Etienne Poirot, and Limin Shen. Actors, Roles and Coordinators - A Coordination Model for Open Distributed and Embedded Systems. COORDINATION '2006, pages 247–265, Berlin, Heidelberg, 2006. Springer-Verlag.

[68] Reo. Reo coordination language. `http://reo.project.cwi.nl/reo/`.

[69] Vinay Setty, Maarten van Steen, Roman Vitenberg, and Spyros Voulgaris. *PolderCast: Fast, Robust, and Scalable Architecture for P2P Topic-Based Pub/Sub*, pages 271–291. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[70] Mary Shaw. Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In *Selected Papers from the Workshop on Studies of Software Design*, ICSE '93, pages 17–32. Springer-Verlag, 1996.

[71] Munindar P. Singh. Information-driven Interaction-oriented Programming: BSPL, the Blindingly Simple Protocol Language. AAMAS '11, pages 491–498, Richland, SC, 2011.

[72] Munindar P. Singh. LoST: Local State Transfer – An Architectural Style for the Distributed Enactment of Business Protocols. In *2011 IEEE International Conference on Web Services*, pages 57–64, July 2011.

[73] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why Do Scala Developers Mix the Actor Model with Other Concurrency Models? ECOOP'13, pages 302–326, Berlin, Heidelberg, 2013. Springer-Verlag.

[74] Guy Theraulaz and Eric Bonbeau. A Brief History of Stigmergy. *Artificial Life*, 5(2):97–116.

[75] Typesafe. Akka Framework. `http://www.akka.io`.

[76] H. Van Dyke Parunak. A Survey of Environments and Mechanisms for Human-Human Stigmergy. In *Proceedings of the 2nd International Conference on Environments for Multi-Agent Systems*, E4MAS'05, pages 163–186, Berlin, Heidelberg, 2006. Springer-Verlag.

[77] Carlos Varela and Gul Agha. Programming Dynamically Reconfigurable Open Systems with SALSA. *SIGPLAN Notices*, 36(12):20–34, December 2001.

[78] Steve Vinoski. Corba: Integrating Diverse Applications within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 35(2):46–55, Feb 1997.

[79] Ken Wakita. First Class Continuation Facilities in Concurrent Programming Language Harmony/2. In *Theory and Practice of Parallel Programming*, LNCS, pages 300–319. Springer-Verlag, 1995.

[80] David W Walker and Jack J Dongarra. MPI: a Standard Message Passing Interface. *Supercomputer*, 12:56–68, 1996.

[81] Peter Wegner. Why Interaction is More Powerful Than Algorithms. *Commun. ACM*, 40(5):80–91, May 1997.

[82] Danny Weyns, Andrea Omicini, and James Odell. Environment as a First Class Abstraction in Multiagent Systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30, 2007.

[83] Xi Wu, Ximeng Li, Alberto Lluch Lafuente, Flemming Nielson, and Hanne Riis Nielson. Klaim-DB: A Modeling Language for Distributed Database Applications. In *International Conference on Coordination Languages and Models*, pages 197–212. Springer, 2015.

Appendix A

Scala Source Code

This appendix only lists Scala code mentioned in this thesis for reference purpose, but not includes all the source code developed for this project.

A.1 `CSL` class

```
1  package agents.envelope
2
3  import akka.actor.{ Actor, ActorRef, ActorContext, Props }
4  import scala.collection.mutable.Map // we are using mutable map
5  import agents.envelope.behavior.Forwarder
6  import agents.envelope.communication.Dummy
7
8  trait CSL {
9    /*
10    * get and set attributes
11    */
12   def set(attributes: Map[String, Any], attrName: String, v: Any) = {
13     attributes.update(attrName, v)
14   }
15   def attr(attributes: Map[String, Any], attrName: String) = {
16     attributes(attrName)
17   }
18
19   def createHandler(context: ActorContext, com: Communication, bhv: Behavior) = {
20     val agent = context.actorOf(Props(new Handler(com, bhv)))
21     com.addHandlers(List(agent))
22     agent
23   }
24
25   def createOutlet(context: ActorContext, com: Communication, otype: String, bhv: Behavior) = {
26     val agent = context.actorOf(Props(new Outlet(bhv)))
27     if (otype == "input") {
28       com.addInlets(List(agent))
29     } else if (otype == "output") {
30       com.addOutlets(List(agent))
31     }
32     agent
33   }
34
35   // change outlet's behavior to bhv
36   def change(handler:ActorRef, outlet: ActorRef, bhv: Behavior) = {
37     outlet.tell(Behv(bhv), handler)
38   }
39
40   // subscribe a trigger
41   def subscribe(trigger:ActorRef, inlet: ActorRef) = {
42     trigger ! inlet
43   }
44
45   /*
46    * communication operations
47    */
48   def tell(sender: ActorRef, recipient: ActorRef, msg: Any) = {
49     recipient.tell(msg, sender)
50   }
51
52   def tellm(sender: ActorRef, recipients: List[ActorRef], msg: Any) = {
53     recipients.foreach(tell(sender, _, msg))
54   }
55
56   def send(recipient: ActorRef, msg: Any) = {
57     recipient ! msg
58   }
59
60   def sendm(recipients: List[ActorRef], msg: Any) = {
61     recipients.foreach(send(_, msg))
62   }
63
```

```
64  def multiSend(recipients: List[ActorRef], data: List[Any]) = {
65    for ((r, d) <- (recipients zip data)) yield send(r, d)
66  }
67
68  def multiTell(sender: ActorRef, recipients: List[ActorRef], data: List[Any]) = {
69    for ((r, d) <- (recipients zip data)) yield tell(sender, r, d)
70  }
71
72  // compose operations
73  def Inputs(context: ActorContext, coms: List[Communication], ins: List[ActorRef], b: Behavior) = {
74    // create a dummy communication
75    val cc = new Dummy(context)
76
77    // create handler with behavior b, which targets ins
78    b.setTargets(ins);
79    val handler = createHandler(context, cc, b)
80    // add the created handler to the dummy communication
81    // we should also add all handlers of all communications to the composed communication
82    var handlers = List[ActorRef]()
83    coms.foreach(c => handlers = handlers ::: c.getHandlers)
84    cc.addHandlers(handlers)
85
86    // create input outlet with behavior Forwarder
87    val input = new Forwarder(context, List(handler))
88    val inlet = createOutlet(context, cc, "input", input)
89
90    // add the created outlet to the dummy communication
91    // we should also add all input and output outlets to the composed communication
92    var inlets = List[ActorRef]()
93    coms.foreach(c => inlets = inlets ::: c.getInlets)
94    // remove merged input outlets
95    inlets = inlets.filterNot(ins.contains(_))
96    cc.addInlets(inlets)
97
98    var outlets = List[ActorRef]()
99    coms.foreach(c => outlets = outlets ::: c.getOutlets)
100   cc.addOutlets(outlets)
101
102   cc
103 }
104
105 def Outputs(context: ActorContext, coms: List[Communication], outs: List[ActorRef],
106             b: Behavior, t: List[ActorRef]) = {
107   // create a dummy communication
108   val cc = new Dummy(context)
109
110   var inlets = List[ActorRef]()
111   coms.foreach(c => inlets = inlets ::: c.getInlets)
112   cc.addInlets(inlets)
113
114   // create output outlet with behavior Forwarder and target t
115   val output = new Forwarder(context, t)
116   val outlet = createOutlet(context, cc, "output", output)
117   // add the created outlet to the dummy communication
118   // we should also add all input and output outlets to the composed communication
119   var outlets = List[ActorRef]()
120   coms.foreach(c => outlets = outlets ::: c.getOutlets)
121   // removed merged output outlets
122   outlets = outlets.filterNot(outs.contains(_))
123   cc.addOutlets(outlets)
124
125   // create handler with behavior b
126   b.setTargets(List(outlet));
127   val handler = createHandler(context, cc, b)
128   // add the created handler to the dummy communication
129   // we should add all handlers of composing communications to the composed communication
130   var handlers = List[ActorRef]()
131   coms.foreach(c => handlers = handlers ::: c.getHandlers)
132   cc.addHandlers(handlers)
133
134   // change the target of composed outlets
135   outs.foreach(_ ! Targets(List(handler)))
136
137   cc
138 }
139
140 /**
141  * binding: (output, input, behavior)
142  */
143 def OutIn(context: ActorContext, coms: List[Communication],
144           bindings: List[(ActorRef, ActorRef, Behavior)]) = {
```

```
145     var ins = List[ActorRef]()
146     var outs = List[ActorRef]()
147     val cc = new Dummy(context)
148
149     bindings.foreach(b => {
150       // set the target of the handler's behavior to the input outlet
151       b._3.setTargets(List(b._2))
152       val handler = createHandler(context, cc, b._3)
153       b._1 ! Targets(List(handler))
154       // merged input outlets
155       ins = ins ::: List(b._2)
156       // merged output outlets
157       outs = outs ::: List(b._1)
158     })
159
160     // add all handlers of all communications
161     // to newly created handlers
162     var handlers = List[ActorRef]()
163     coms.foreach(c => handlers = handlers ::: c.getHandlers)
164     cc.addHandlers(handlers)
165
166     // get all input outlets of all communications
167     // remove merged input outlets in the binding
168     var inlets = List[ActorRef]()
169     coms.foreach(c => inlets = inlets ::: c.getInlets)
170     inlets = inlets.filterNot(ins.contains(_))
171     cc.addInlets(inlets)
172
173     // get all output outlets of all communications
174     // remove merged output outlets in the binding
175     var outlets = List[ActorRef]()
176     coms.foreach(c => outlets = outlets ::: c.getOutlets)
177     outlets = outlets.filterNot(outs.contains(_))
178     cc.addOutlets(outlets)
179
180     cc
181   }
182
183   def delay(miliseconds: Int) = {
184     Thread.sleep(miliseconds)
185   }
186
187   /*
188    * List operations
189    * A list is a Scala list
190    */
191   def add(elem: Any, l: List[Any]) = {
192     elem :: l
193   }
194
195   def append(elem: Any, l: List[Any]) = {
196     l ::: List(elem)
197   }
198
199   /*
200    * index starts at 0
201    */
202   def get(l: List[Any], i: Int): Any = {
203     l(i)
204   }
205   def remove(l: List[Any], i: Int): List[Any] = {
206     l.take(i - 1) ::: l.drop(i - 1).tail
207   }
208   /*
209    * check whether the list is empty
210    */
211   def isEmpty(l: List[Any]): Boolean = {
212     l.isEmpty
213   }
214
215   def empty() = {
216     List[Any]()
217   }
218
219   /*
220    * return the size of the list
221    */
222   def size(l: List[Any]): Int = {
223     l.length
224   }
225 }
```

A.2 MOM2M2

```
package agents.envelope.communication

import akka.actor.{ ActorRef, ActorContext }
import agents.envelope.Communication
import agents.envelope.behavior.Forwarder
import agents.envelope.behavior.Aggregator

class Mom2m2(ct: ActorContext) extends Communication(ct) {
  var recipients: List[ActorRef] = null;
  var participants: List[ActorRef] = null;

  var cond: List[Any] => Boolean = null
  var aggr: List[Any] => Any = null

  var outlet: ActorRef = null
  var inlet: ActorRef = null
  var handler: ActorRef = null

  def launch() = {
    println("inlet: " + inlet)
    sendm(participants, inlet);
  }
  def init() = {
    recipients = attr(attributes, "recipients").asInstanceOf[List[ActorRef]]
    participants = attr(attributes, "participants").asInstanceOf[List[ActorRef]]
    cond = attr(attributes, "cond").asInstanceOf[List[Any] => Boolean]
    aggr = attr(attributes, "aggr").asInstanceOf[List[Any] => Any]

    // create output outlet
    val output = new Forwarder(context, recipients)
    outlet = createOutlet(context, this, "output", output)

    // create handler
    val aggregator = new Aggregator(context, List(outlet), cond, aggr)
    handler = createOutlet(context, this, "output", aggregator)

    // create input outlet
    val input = new Forwarder(context, List(handler))
    inlet = createOutlet(context, this, "input", input)

  }
}
```

A.3 Using SOM2M

```
package agents.envelope.example.som2m

import akka.actor.{ ActorRef, Actor, ActorSystem, Props }
import agents.envelope.communication.Som2m

import scala.collection.mutable.Map

// To run:
// sbt "run-main agents.envelope.example.som2m.Test 3 4 4"
// sbt "run-main agents.envelope.example.som2m.Test 4 3 4"
// yes no required
object Test {
  def main(args: Array[String]) {
    val system = ActorSystem("TEST")
    val ya = args(0).toInt // # of yes actors
    val na = args(1).toInt // # of no actors
    val required = args(2).toInt // # of required
    // Create a sender and a receiver based on EndPoints
    val tester = system.actorOf(Props[MainTestActor])
    tester ! SOM2MStart(ya, na, required)
  }
}

class MainTestActor extends Actor {
  def receive = {
    case SOM2MStart(ya, na, required) => {
      val s = context.actorOf(Props[MyActor], "sender")
      val r = context.actorOf(Props[MyActor], "receiver")
      // Create multiple participants
      val yal = (for (i <- 1 to ya) yield context.actorOf(Props[YesPart])).toList
```

```
31        val nal = (for (i <- 1 to na) yield context.actorOf(Props[NoPart])).toList
32        var parts = yal ::: nal
33        var cond = funs.cond(ya + na, required, _: List[Any])
34        var aggr = funs.aggr(required, _: List[Any])
35
36        val som2m = new Som2m(context)
37        som2m.setAttr(Map(
38          "recipients" -> List(r),
39          "content" -> "go through",
40          "query" -> "voting",
41          "parts" -> parts,
42          "cond" -> cond,
43          "aggr" -> aggr))
44        som2m.launch()
45      }
46      case m => println(m)
47    }
48  }
49
50  // recipient actor
51  class MyActor extends Actor {
52    println(self + " created")
53    def receive = {
54      case m      => println("Received " + m + " at " + self)
55    }
56  }
57
58
59  // participants who say 'yes'
60  class YesPart extends Actor {
61    println(self + " created")
62    def receive = {
63      case m => {
64        println("yesyesyes:" + sender)
65        sender ! "yes"
66      }
67    }
68  }
69  // participants who say 'no'
70  class NoPart extends Actor {
71    println(self + " created")
72    def receive = {
73      case m => {
74        println("nonono:" + sender)
75        sender ! "no"
76      }
77    }
78  }
79
80  // cond and aggr functions
81  object funs {
82    // Signature: (Int, Int, Any) => Boolean
83    // @params:
84    //  total:    the total number of participants
85    //  required: the required number of vote "yes"
86    //  l:        the list of responses
87    //  @return:
88    //    true: terminate
89    //    false: no terminate
90    def cond(total: Int, required: Int, l: List[Any]): Boolean = {
91      var na = 0 // no. of agreement
92      var nd = 0 // no. of disagreement
93      l.foreach(x =>
94        if (x == "yes") {
95          na = na + 1
96        } else {
97          nd = nd + 1
98        })
99      println("Required: " + required + " na: " + na)
100     if (na >= required || (nd + required) > total)
101       true
102     else
103       false
104   }
105
106   // Signature: (Int, Int, Any) => Boolean
107   // @params:
108   //  total:    the total number of participants
109   //  required: the required number of vote "yes"
110   //  l:        the list of responses
111   //  @return:
```

```
112   //    true:  succeed
113   //    false: fail
114   def aggr(required: Int, l: List[Any]): Boolean = {
115     var na = 0 // no. of agreement
116     l.asInstanceOf[List[String]].foreach(x => if (x == "yes") na = na + 1)
117     if (na >= required) true else false
118   }
119 }
120
121 case class SOM2MStart(val ya: Int, val na: Int, val required: Int)
```

A.4 Implementing and Testing Barrier Using MOM2M

```
1  package agents.envelope.example.barrier
2
3  import akka.actor.{ ActorRef, Actor, ActorSystem, Props }
4  import scala.collection.mutable.Map
5  import agents.envelope.communication.Mom2m
6
7  // To run:
8  // sbt "run-main agents.envelope.example.barrier.Test 2 10"
9  // where 2 is the number of processes to be synchronized
10 //       10 is the number iterations
11 object Test {
12   def main(args: Array[String]) {
13     val system = ActorSystem("MOM2M")
14     val tester = system.actorOf(Props(new Tester()))
15     tester ! Ini(args(0).toInt, args(1).toInt)
16   }
17 }
18
19 class Tester extends Actor {
20   def receive = {
21     case Ini(arg0, arg1) => {
22       val processes = (for (i <- 1 to arg0) yield context.actorOf(Props(new SyncActor(arg1)))).toList
23
24       var cond = funs.cond(arg0, _: List[Any])
25       var aggr = funs.aggr(_: List[Any])
26       val mom2m = new Mom2m(context)
27       mom2m.setAttr(Map("recipients" -> processes,
28         "participants" -> processes,
29         "cond" -> cond, "aggr" -> aggr))
30       mom2m.launch()
31     }
32   }
33 }
34
35 case class Ini(val par: Int, val iteration: Int)
36 case class Start(val inlet: ActorRef)
37
38 class SyncActor(val it: Int) extends Actor {
39   println(self + " created")
40   var nit = it
41   var input: ActorRef = null
42
43   def receive = {
44     case "go" => {
45       if (nit > 0) {
46         println("Finish iteration: " + nit + " " + self);
47         nit = nit - 1
48         input ! "Iteration " + nit
49         println("Start iteration " + nit + " " + self)
50       }
51     }
52     case m => {
53       if (m.isInstanceOf[ActorRef]) {
54         input = m.asInstanceOf[ActorRef]
55         println("Start iteration " + nit + " " + self)
56         input ! "Iteration " + nit
57       }
58     }
59   }
60 }
61
62 object funs {
63   // @params:
64   //  total:    the total number of participants
65   //  l:        the list of responses
66   //  @return:
```

```
67    //    true: terminate
68    //    false: no terminate
69    def cond(total: Int, l: List[Any]): Boolean = {
70      if (l.length == total)
71        true
72      else
73        false
74    }
75    // @params:
76    //  l:        the list of responses
77    //  @return:
78    //     "go" message
79    def aggr(l: List[Any]): Any = {
80      "go"
81    }
82  }
```

A.5 Output Merge Example

```
1  package agents.envelope.communication
2
3  import akka.actor.{ ActorRef, ActorContext }
4  import agents.envelope.Communication
5
6  import scala.collection.mutable.Map
7  import agents.envelope.behavior.Forwarder
8
9  class OutMerge(ct: ActorContext) extends Communication(ct) {
10   val c1 = new Broadcaster(ct)
11   val c2 = new Broadcaster(ct)
12   val c3 = new Broadcaster(ct)
13
14   var senders1: List[ActorRef] = null
15   var senders2: List[ActorRef] = null
16   var senders3: List[ActorRef] = null
17   var recipients: List[ActorRef] = null
18
19   def init() = {
20     senders1 = attr(attributes, "senders1").asInstanceOf[List[ActorRef]]
21     senders2 = attr(attributes, "senders2").asInstanceOf[List[ActorRef]]
22     senders3 = attr(attributes, "senders3").asInstanceOf[List[ActorRef]]
23     recipients = attr(attributes, "recipients").asInstanceOf[List[ActorRef]]
24
25   }
26   def launch() = {
27     c1.setAttr(Map("recipients" -> null, "participants" -> senders1))
28     c2.setAttr(Map("recipients" -> null, "participants" -> senders3))
29     c3.setAttr(Map("recipients" -> null, "participants" -> senders3))
30
31     val outset = List(c1.out, c2.out, c3.out)
32
33     val b = new Forwarder(ct, null)
34
35     val cc = Outputs(context, List(c1, c2, c3), outset, b, recipients)
36     addInlets(cc.getInlets)
37     addHandlers(cc.getHandlers)
38     addOutlets(cc.getOutlets)
39
40     // senders1 still use c1's input outlets to send messages
41     // because c1's input outlets belong to the composed communication
42     sendm(senders1, c1.getInlets)
43     sendm(senders2, c2.getInlets)
44     sendm(senders3, c3.getInlets)
45   }
46
47 }
```

A.6 Output-input Merge Example

```
1  package agents.envelope.communication
2
3  import akka.actor.{ ActorRef, ActorContext }
4  import agents.envelope.Communication
5  import agents.envelope.Behavior
6
7  import agents.envelope.behavior.Forwarder
```

```
8  import agents.envelope.behavior.Selector
9  import agents.envelope.behavior.Aggregator
10 import scala.collection.mutable.Map
11
12 /*
13  * This class demonstrates output-input merge.
14  * Specifically, it merges two broadcaster communications,
15  * each has one input outlet and one output outlet.
16  * The two communications are connected by a handler having
17  * forwarder behavior (defined in bindings).
18  */
19 class OutputInputMerge(ct: ActorContext) extends Communication(ct) {
20   var workers: List[ActorRef] = null
21
22   val c1 = new Broadcaster(ct)
23   val c2 = new Broadcaster(ct)
24   var senders: List[ActorRef] = null
25   var recipients: List[ActorRef] = null
26
27   def init() = {
28     senders = attr(attributes, "senders").asInstanceOf[List[ActorRef]]
29     recipients = attr(attributes, "recipients").asInstanceOf[List[ActorRef]]
30   }
31
32   def launch() = {
33     // attributes set to null because for c1, recipients is not relevant
34     // for c2, participants is not relevant
35     c1.setAttr(Map("recipients" -> null, "participants" -> senders))
36     c2.setAttr(Map("recipients" -> recipients, "participants" -> null))
37
38     val bindings = List((c1.out, c2.in, new Forwarder(context, List(c2.in))))
39
40     val cc = OutIn(ct, List(c1, c2), bindings)
41     addInlets(cc.getInlets)
42     addHandlers(cc.getHandlers)
43     addOutlets(cc.getOutlets)
44
45     sendm(senders, getInlets)
46   }
47 }
```

A.7 Router2

```
1  package agents.envelope.communication
2
3  import akka.actor.{ActorRef,ActorContext}
4
5  import agents.envelope.Communication
6  import agents.envelope.behavior.Forwarder
7  import agents.envelope.behavior.Selector
8
9  class Router2(ct: ActorContext) extends Communication(ct) {
10   var recipients: List[ActorRef] = null;
11   var parts: List[ActorRef] = null
12   var select: (Any, List[ActorRef]) => List[ActorRef] = null;
13
14   var outlet: ActorRef = null
15
16   def launch() = {
17     sendm(parts, outlet);
18   }
19
20   def init() = {
21     recipients = attr(attributes, "recipients").asInstanceOf[List[ActorRef]]
22     parts = attr(attributes, "parts").asInstanceOf[List[ActorRef]]
23     select = attr(attributes, "select").asInstanceOf[(Any, List[ActorRef]) => List[ActorRef]]
24
25     // create the output outlet
26     val output = new Selector(context, recipients, select)
27     outlet = createOutlet(context, this, "output", output)
28   }
29 }
```