Describing and Supporting Complex Interactions in Distributed Systems

Andrew Berry
Department of Computer Science and Electrical Engineering
The University of Queensland
<andyb@dstc.edu.au>

26 July 2001

Abstract

The idea of building software that spans many computers has been a significant driver for research since the earliest days of computing. Despite this past effort, there are still considerable resources being applied to the problems of building distributed software systems. This thesis proposes an alternative approach to building distributed systems that is both pragmatic and has a sound theoretical basis. The approach is driven by the following key assertions:

1. The construction of software systems has become an evolutionary rather than revolutionary process. New software must extend or incorporate old software.
2. Distributed applications must often coordinate the activities of multiple participants with varying relationships. Static abstractions like client-server are too primitive and inflexible to describe such relationships.
3. Communication networks cannot consistently deliver high-bandwidth, low-latency, low-failure communications. Distributed software must be able to deal with the degradation or loss of communication.

Given this context, this thesis describes Finesse, a system for describing and supporting the complex interactions of components in distributed applications. Finesse uses the notion of event relationships to describe the visible behaviour of components and how those behaviours are coordinated to satisfy the requirements of an application. There are three significant aspects of Finesse, each of which makes a novel contribution to the body of research:

1. A semantic model that provides the theoretical basis for event relationships.
2. A programming language for describing component interfaces and the relationships between events occurring on those interfaces.
3. A distributed runtime engine that manages and mediates communication between components to implement distributed programs described using the Finesse programming language.

The system implies a mediated communication architecture, where a mediating component accepts events from components and distributes appropriate notifications to other components. This decoupling of components and the mediation semantics allow the necessary transformation of data and event correlation to support legacy applications. The event relationship semantics, however, also allow the mediator to be fully distributed and asynchronous. This asynchrony allows us to deal with high-latency, high-failure communication networks. The design of the programming language allows us to develop mediating 'components' that can be nested within a higher level mediator, thus giving us the ability to build a flexible library of abstractions for relationships between participating components.

The thesis includes a set of examples that demonstrate the approach and its strengths. Through the examples and the arguments expressed in the work, this thesis demonstrates the power and utility of the event relationship approach to building distributed systems.

Statement of Sources

I, Andrew Berry, declare that the work presented in this thesis is, to the best of my knowledge and belief, original, except as acknowledged in the text, and that the material has not been submitted, either in whole or in part, for a degree at this or any other university.

Acknowledgments

I want to thank a number of people and organizations who have supported, directed, and assisted me in completing this thesis. My supervisor, Professor Simon Kaplan, provided a research environment full of inspirational, supportive, people and gave me the freedom to follow my gut feelings and explore with abandon. I owe him many thanks for his support and belief in me. Colin Fidge was my original motivator, arousing my interest in the unique problems of distributed systems some years before I began this journey. I owe him many thanks for that and for the numerous subsequent discussions that helped to direct my thesis and give me confidence in my findings. I would also like to thank the staff and management of the DSTC for their support and encouragement throughout my thesis. In particular, Kerry Raymond was crucial in directing my thoughts along the paths that led to this thesis result. David Garlan hosted me during a stimulating six-month visit to Carnegie Mellon University in 1998, adding another dimension to my understanding of the related work. My current employers, ADC Software Systems Australia, have been very sympathetic while I have struggled to complete the thesis part-time during the last two years. I would especially like to thank my manager David Pope for understanding my needs and allowing me to take leave whenever necessary to progress my work.

The thesis was begun in 1995 at a particularly difficult time in my life. I had not long before lost my young son to a congenital illness and was struggling to find my emotional feet. In retrospect, I think that beginning a thesis was the right decision for me, but at the time I had many doubts and the first year was particularly difficult. Now that I am at the end of the thesis, I can look back and say without reservation that it has been an experience I will treasure, as much for the relationships I have developed as for the academic and professional achievements. Given this context, I most want to thank the friends and family who have helped me through the process of finding life and learning to love again. Without them, I don't think the thesis would have been completed.

Contents

1 Introduction **13**
1.1 Pervasive Software 13
1.2 The Discipline of Distributed Systems 14
1.3 A Plethora of Communication Technologies 15
1.4 Introducing *Finesse* 16

2 System Overview **19**
2.1 Influences 20
2.1.1 The A1$\sqrt{}$ Model and RM-ODP 20
2.1.2 Architecture and Coordination Languages 20
2.2 Architectural Model 21
2.3 Behavioural Model 22
2.4 Language 22
2.4.1 Parameterizable RPC 23
2.4.2 Example: File Access using RPC 24
2.4.3 Multicast RPC 25
2.4.4 Replicated File Access 26
2.5 Runtime Infrastructure 27
2.6 Concluding Remarks 28

3 Literature Review **29**
3.1 Software Architecture 29
3.2 Distributed Systems Architecture 31
3.2.1 Reference Model of Open Distributed Processing 32
3.2.2 The A1$\sqrt{}$ Architecture Model 32
3.3 Coordination 33
3.4 Component Systems 35
3.5 Interaction Paradigms for Distributed Systems 36
3.5.1 Remote Procedure Call 36
3.5.2 Message Passing 37
3.5.3 Multicast 38
3.5.4 Streams 38
3.5.5 Replication 39
3.5.6 Distributed Shared Memory 39
3.5.7 Distributed File Systems 39
3.5.8 Transactions 40

3.5.9 Event-based Interaction . . . 41
3.5.10 Connection-based Interaction . . . 41
3.5.11 Intelligent Agents . . . 42
3.5.12 Mobile Agents . . . 42
3.6 Distributed Systems Infrastructure . . . 43
3.6.1 Commercial Environments . . . 44
3.6.2 Research Prototypes . . . 45
3.7 Computer Supported Cooperative Work . . . 46
3.7.1 CSCW Systems and Sociological Theory . . . 47
3.7.2 CSCW Toolkits . . . 48
3.7.3 CSCW Criticism of Existing Distributed Systems . . . 48
3.7.4 Languages for Describing Collaboration . . . 49
3.8 A Scaffolding supporting Finesse . . . 50

4 Semantics of Behavioural Model **51**
4.1 Base Execution Model . . . 52
4.2 Templates describe Programs . . . 54
4.3 Program Execution . . . 56
4.4 Parameters . . . 57
4.4.1 Defining Parameter Relationships . . . 57
4.4.2 Identifying Event Parameters . . . 58
4.5 Guards and Timing Constraints . . . 59
4.6 Distributing the Execution . . . 60
4.6.1 Synchronization . . . 63
4.6.2 Autocratic Choice . . . 63
4.6.3 Safe Programs . . . 63
4.6.4 Optimistic Execution . . . 64
4.7 The Significance of Location . . . 65
4.8 Formal Specification . . . 66
4.9 Introducing Z . . . 67
4.10 Basic Behaviour Description . . . 68
4.10.1 Base Types . . . 69
4.10.2 Events and Event Templates . . . 70
4.10.3 Causality Graphs . . . 71
4.10.4 Guards and Time Semantics . . . 72
4.10.5 Correct Execution . . . 73
4.10.6 Transition Semantics . . . 75
4.11 Formalising Distribution . . . 77
4.11.1 Templates and Locations . . . 78
4.11.2 Initialization . . . 79
4.11.3 Distributed State . . . 79
4.11.4 Distributed Transitions . . . 80
4.11.5 History Updates . . . 82
4.11.6 Properties of Transitions . . . 82
4.12 Concluding Remarks . . . 83
4.13 Acknowledgments . . . 84

5 Language **85**

5.1 Introduction . . . 85
5.2 Basic Syntax and Structure . . . 86
5.2.1 Structure of a *Finesse* Program . . . 86
5.2.2 Describing Roles . . . 87
5.2.3 Describing Interactions . . . 87
5.2.4 Event Templates . . . 88
5.2.5 Named Behaviours . . . 89
5.3 Describing Behaviour . . . 90
5.3.1 Introducing the Causality Operator . . . 90
5.3.2 Complex Expressions . . . 90
5.3.3 Logical AND . . . 91
5.3.4 Logical OR . . . 92
5.3.5 Exclusive OR . . . 93
5.3.6 Combining Logical Operators and Complex Expressions . . . 93
5.3.7 The Logic of Cardinality Constraints . . . 94
5.4 Control Flow . . . 96
5.4.1 Roles, Interactions and Iteration . . . 96
5.4.2 Iteration . . . 97
5.5 Resolving Event References . . . 98
5.6 Guards . . . 99
5.6.1 Time Guards . . . 100
5.6.2 Event Causality Guards . . . 100
5.7 Parameter Relationships . . . 101
5.8 Reuse and Generics . . . 102
5.9 Concluding Remarks . . . 103

6 Runtime Engine **105**

6.1 Overview . . . 105
6.2 Representing Programs . . . 107
6.3 Initialising a Program Execution . . . 107
6.4 Executing a Program . . . 109
6.4.1 Causal Enablement . . . 110
6.4.2 Guards and Timing Relationships . . . 110
6.4.3 Enabling an Event . . . 111
6.4.4 Managing the Event History . . . 112
6.4.5 Parameter Relationships . . . 114
6.5 Language Binding . . . 115
6.6 Other Design Issues . . . 116
6.6.1 Garbage Collection . . . 116
6.6.2 Error Detection and Handling . . . 116
6.6.3 Reliable Communication . . . 117
6.7 Concluding Remarks . . . 118

7 End-to-End Example **119**

7.1 Example Program . . . 119
7.2 Compilation . . . 120
7.2.1 Roles Compilation . . . 120
7.2.2 Interactions Compilation . . . 121
7.3 Participant Behaviour . . . 122
7.4 Instantiation . . . 123
7.5 Execution . . . 123
7.5.1 Initial Client State . . . 124
7.5.2 Initial Server State . . . 124
7.5.3 Client Executes Send . . . 124
7.5.4 Server Receives Client Send Notification . . . 125
7.5.5 Server Executes Receive . . . 126
7.5.6 Server Executes Send . . . 126
7.5.7 Client Receives Server Send Notification . . . 127
7.5.8 Client Executes Receive . . . 128
7.6 Concluding Remarks . . . 129

8 Examples **131**

8.1 Programmable, Component-Oriented Middleware . . . 132
8.1.1 Reliable Multicast . . . 132
8.1.2 Two-phase Commit . . . 133
8.1.3 Streaming Data . . . 135
8.2 Enterprise Application Integration . . . 140
8.3 Business to Business Interaction . . . 142
8.4 Computer Supported Cooperative Work . . . 144
8.5 Concluding Remarks . . . 147

9 Discussion and Related Work **149**

9.1 Does it Satisfy? . . . 149
9.2 A Critical Examination . . . 151
9.2.1 Complexity . . . 151
9.2.2 Finesse Language Syntax . . . 152
9.2.3 Quality of Service . . . 153
9.2.4 Dynamic Behaviour Instantiation . . . 154
9.2.5 Security . . . 154
9.3 Related Work . . . 156
9.3.1 Models for Parallel and Distributed Systems . . . 156
9.3.2 Mobile Agent Technology . . . 157
9.3.3 Coordination and Architecture Description Languages . . . 159
9.3.4 Middleware Platforms . . . 160
9.3.5 EAI Platforms . . . 161
9.3.6 B2B Platforms . . . 162
9.4 Concluding Remarks . . . 162

10 Conclusion **163**

Chapter 1

Introduction

This thesis describes a new approach to the construction of distributed systems. The approach suggests building distributed systems by declaratively specifying the relationships between the visible behaviours of participating components, and executing this specification on a distributed, asynchronous, execution engine. The approach and the system that implements it is called *Finesse*. *Finesse* is intended to deal with the realities of software construction and use: while innovative and theoretically sound, it addresses pragmatic concerns of the software engineering discipline.

In this introductory text, a set of assertions about the nature of software construction and distributed systems is introduced. After describing *Finesse* in subsequent chapters of the thesis, the approach will be evaluated against these assertions.

1.1 Pervasive Software

Software running on computers has become a pervasive part of life in our society: The delivery of fuel and ignition timing in a car engine is controlled by software; Banks and commodity markets are entirely dependent on computerized transaction systems; And many electrical appliances in our homes are now controlled by software. Almost every office desk has a computer, and the proportion of homes with a personal computer and an Internet connection is growing rapidly.

Witness also the worldwide flurry of activity associated with the millennium bug. The significance of the millennium bug highlights our dependence on software, is a

reflection of the long-term value of existing software, and is an indication that software systems must be built to deal with changing environments.

These two facts—the pervasiveness of software and intrinsic value of existing software—lead us to a first assertion about software construction:

> *The construction of software has become an evolutionary rather than revolutionary process. New software must extend or incorporate old software.*

1.2 The Discipline of Distributed Systems

The notion that software can be built as a set of independent but cooperating components running on distinct processors has been a driver for research for more than two decades. Consider the early work of Lamport[77] on distributed clocks, the Cambridge distributed system[132], or Gifford's work on replication[49]. This field of research is traditionally known as distributed systems. Considerable resources have been applied to the many problems associated with distributed systems, and there have been many successes.

The adoption of distributed systems in commercial settings has been steadily increasing. This is driven by the steady migration of many business functions to desktop computers, and the increasing need to support those business functions with timely and relevant information in widely dispersed organizations. There has also been a push towards integrating once-distinct systems within an organization to provide a single, coherent view of the organization on the desktop, and to use computer systems as the basis for cooperative work between physically distributed users. These applications are distributed by necessity, rather than to satisfy performance and other technical objectives.

A key feature of infrastructures that successfully support existing distributed systems is appropriate abstractions for connecting system components. Remote procedure call[15], distributed transactions[136], and 3-tier architectures[37] are all abstractions that have been used successfully. These abstractions tend to be static, programmatic, relatively low-level, and oriented towards two-party interaction. The emergence of

multimedia, computer supported cooperative work (CSCW), and the integration of multiple sources of information suggest the need for more flexible and higher-level abstractions. The second assertion of this thesis is thus:

> *Distributed applications must coordinate the activities of multiple participants with varying relationships. Static abstractions like client-server are too primitive and inflexible to describe such relationships.*

1.3 A Plethora of Communication Technologies

While distributed systems have been evolving through the 90s, communication and telephony systems have been exploding. Mobile phones have become a ubiquitous tool of the increasingly-mobile business worker, and the Internet has become the preferred way of sending and receiving information for both personal and business purposes. A key aspect of technologies in these areas is that they are international: communication and cooperation across national boundaries is now both common and relatively inexpensive.

This explosion of communication technologies has emphasized, however, the shortcomings of ubiquitous networking and communications. While the Internet as a whole is highly reliable, the failure of a single node can result in the loss of communication between two entities and such failures occur relatively often. The significance of international communication is that the latency of communication is theoretically limited by the speed of light in a vacuum ($c = 3x10^8 m/s$): this latency becomes a significant performance limitation on synchronous interactions. For example, a round-trip communication between nodes 10,000km apart has a latency of $(10,000,000/c) \times 2$ or $60ms$ if communication occurs at the maximum theoretical speed. This time is many orders of magnitude higher than the time taken to execute a processor instruction or local memory fetch.

A further complication is that the coverage offered by mobile communication networks is limited to areas of high-density population because of costs, and that both the bandwidth and reliability of such networks is limited. This leads to a third assertion of this thesis:

Communication networks cannot consistently deliver high-bandwidth, low-latency, low-failure communications. Distributed software must be able to deal with high latency and the degradation or loss of communication.

1.4 Introducing *Finesse*

With the assertions of the preceding sections in mind, this thesis proposes the *Finesse* approach to the construction of distributed systems. The thesis describes the approach and a system built to support the approach, and shows its utility through examples. There are three components of *Finesse*, each of which contributes to the body of research in distributed systems:

1. A semantic model that provides the theoretical basis for describing relationships between components and executing programs to realize those relationships.

2. A programming language for describing the visible behaviour of components and the relationship between the behaviours that results in the desired distributed application.

3. A distributed runtime engine that manages and mediates communication between components to implement distributed programs described in the terms of the semantic model.

This approach can integrate existing software, provide a basis for building flexible and high-level abstractions, and places minimal constraints on the infrastructure used to realize an application.

The following chapter provides an overview of the *Finesse* system to give readers a high-level understanding of the components and goals of *Finesse*. Chapter 3 surveys relevant literature to define the scaffolding upon which the ideas of this thesis rest. Chapters 4, 5, and 6 describe the execution model, programming language, and implementation of *Finesse* respectively. Chapter 7 provides an end-to-end programming example to illustrate the concepts in the preceding chapters and their relationships. Chapter 8 demonstrates the capabilities and wide applicability of the approach through

examples and associated discussion. Chapter 9 evaluates the *Finesse* system by returning to the key assertions of this chapter to show that *Finesse* addresses these assertions, discussing the strengths and weaknesses of the approach, and showing the novelty of *Finesse* through comparison with related work. In conclusion, the thesis emphasizes the contribution of this work to the practice of distributed software construction.

Chapter 2

System Overview

This chapter gives an informal overview of *Finesse* with the intention of preparing readers for the more detailed discussions in subsequent chapters. It describes the primary influences, architecture model, and key components of the approach, including some simple examples to illustrate the concepts.

Finesse is a platform and an approach for describing and supporting complex interactions between components in distributed systems. The system comprises an executable semantic model, programming language, and a runtime infrastructure for executing programs. *Finesse* began as an attempt to realize the architectural concepts of the A1$\surd$ model[11] in a development environment for distributed systems. The initial focus was on a prototype language, with the goal of investigating programming models that allowed description of arbitrary interaction mechanisms within the architectural framework of A1$\surd$. A secondary goal was to use a behavioural model that allowed components to execute autonomously without the need for a central mediator, since both A1$\surd$ and RM-ODP are intended to support interactions that cross enterprise boundaries. In these situations a central mediator is undesirable.

Through a process of revision and refinement, the prototype language has become the *Finesse* language syntax described in this thesis. The underlying behavioural model was formalized after it was shown that the language could describe complex interactions in a flexible and modular fashion. A prototype implementation capturing the semantic model and the architectural principles embodied in the language was built concurrently with the formalism to show the feasibility of the language and model.

The goal of autonomy for the distributed components has been realized through careful design of the behavioural model: it allows behaviour to be arbitrarily distributed across a set of autonomous runtime engines communicating only through asynchronous messaging.

2.1 Influences

2.1.1 The A1$\surd$ Model and RM-ODP

Finesse was developed to support the principles and concepts of the A1$\surd$ architecture model for distributed systems. It directly uses the notions of binding, interface and role defined by the model (described in section 2.2), and early work on the behavioural model[102] was performed in conjunction with researchers from the CRC for Distributed Systems Technology where the model was developed.

The A1$\surd$ model has a strong relationship with the ISO Basic Reference Model of Open Distributed Processing[63, 105]. The notions of binding and interface are strongly related to similar ODP concepts, although the A1$\surd$ model does not use the ODP viewpoints. *Finesse* benefits from the open systems approach in not prescribing specific data models or infrastructure, although the prototype implementation uses a particular language and network infrastructure.

2.1.2 Architecture and Coordination Languages

Finesse is also strongly influenced by work in architecture description languages. Research into software architecture [117, 118] supports a model of programming that distinguishes software components and their connectors. This model promotes reuse and reduces the coupling of software components, and a number of architecture description languages have been developed, for example Wright[45] and Rapide[80]. The primary difference between *Finesse* and these languages is that architecture description languages are typically oriented towards simulation and analysis of architectures rather than building software systems.

Recent efforts in developing coordination languages and models[31] for distributed systems have focused on the need to distinguish components and their coupling, and

incorporate strong abstraction capabilities in languages for programming distributed systems. These same principles are used in *Finesse*, although there are a number of differences between *Finesse* and these systems, discussed further in chapter 9. These languages and models are usually executable and intended for building software systems.

2.2 Architectural Model

The primary concept in *Finesse* is the *binding* as defined in the A1$\sqrt{}$ model, which is an entity that encapsulates the communication between distributed components participating in an application. A binding is equivalent to the notion of a *connector*, a term commonly found in software architecture literature[1, 117, 106]. Bindings are described in terms of the following fundamental concepts:

role: a binding has a set of roles that can or must be filled by participating components. One or more components can fulfill a single role, providing a convenient abstraction for group interaction.

interface: components have interfaces through which they interact with their environment. Each interface is assigned to one or more roles in the binding and must implement the behaviour specified for those roles.

events: components participate in a binding (interact) by executing events at their interfaces. Events are immutable and have a location (interface), parameters, and an execution time.

event relationships: event relationships specify the behaviour and interactions of a binding by describing the relationships between events occurring at component interfaces.

A binding is instantiated by nominating a *Finesse* program or some compiled form of that program, and a set of components to fulfill the roles of the binding. The underlying distributed infrastructure is required to establish an appropriate set of network connections and supporting components to implement the *Finesse* program. A *Finesse*

program also could be used to generate stubs for the participating components in a similar manner to CORBA IDL, meaning that *Finesse* is somewhat independent of the language used to build the participating components.

2.3 Behavioural Model

Event relationships provide the basis for describing behaviour in bindings. Event relationships capture the dependencies between events occurring at the interfaces of components participating in a distributed application. Three distinct types of event relationship are identified:

Causal relationships which describe the causal dependencies between events;

Parameter relationships which describe the relationships between parameters of causally related events. Parameter relationships specify the content of messages passed between interacting components in a declarative, application-oriented manner;

Timing relationships which describe the real-time relationship between events. These relationships can be used to describe, for example, timeouts or quality of service requirements of interactions.

The causal relationships between events form the basis of the execution model associated with a *Finesse* implementation, and in a given program, these relationships describe the control flow. Parameter and timing relationships declaratively specify the data flow and time-dependent properties of a binding program.

These concepts, combined with the notions of *binding*, *interface* and *role*, provide an extremely powerful technique for the description of distributed systems interaction. For example, it is possible to succinctly describe and easily extend remote procedure call, group communication, and stream behaviour. The declarative specification of data flow also provides a basis for optimization of messaging.

2.4 Language

The *Finesse* language provides a syntax to express the structuring and behaviour of a binding. The syntax is not intended to be the only way of writing such programs, but

provides an illustration of how the behavioural model can be realized in a specification or programming language. It has some similarities with process algebras like CSP and LOTOS, but includes more significant facilities for describing data and uses a true concurrency model.

It is easiest to give a flavour for the *Finesse* language using some examples. The following examples demonstrate the basic features and structuring of the *Finesse* language. We use the language to define RPC interaction, then extend RPC to implement multicast RPC with minimal changes.

2.4.1 Parameterizable RPC

The following binding describes a parameterizable RPC interaction with two roles, client and server. The *Roles* section defines the behaviour of the participants. The *Interactions* section defines the relationship between the roles. A set of required messages and hence appropriate network connections can be derived from the behaviour.

```
Binding RPC {
  -- simple, parameterizable RPC

  Roles {
    -- the client role is parameterized by a set of input and
    -- output values
    Client(IN, OUT) {
       -- the client executes a send (output) followed by a
       -- receive (input)
       send!(IN) -> receive?(OUT)
    }
    -- the server role is similarly parameterized
    Server(IN, OUT) {
       -- the server executes a receive followed by a send
       receive?(IN) -> send!(OUT)
    }
  }
```

```
  Interactions {
     -- the client send causes the server to receive,
     -- with parameters matched by name
     Client.send -> Server.receive {*= prev} AND

     -- the server send causes the client to receive,
     -- with parameters matched by name
     Server.send -> Client.receive {*= prev}
  }
}
```

The following syntactic elements are used:

- $Client(IN, OUT)$ introduces the client role, parameterized by a set of sent values sent (IN) and a set of received values (OUT).
- $send!(IN)$ indicates an event where the client outputs the IN values
- $receive?(OUT)$ indicates an event where the client accepts the OUT values
- $->$ indicates a causal relationships between events, that is $A -> B$ specifies that A affects B hence must occur before B.
- *Client.send* refers to the execution of the client send event.
- $*=$ *prev* indicates that the parameters of the current event should be set equal to parameters having the same name in the previous event (i.e. name equivalence).

2.4.2 Example: File Access using RPC

Use of this parameterizable RPC binding is demonstrated in the following binding definition for file I/O:

```
Binding FileIO {
  -- read-only file access using RPC

  Import RPC;

  Roles {
    -- Client and Server implement open/read/close
    Client {
       open  { RPC.Client ((name:string), (fh:handle)) } ->
       loop {
         read  { RPC.Client ((fh:handle, bytes:int),
                             (buf:buffer,bytes:int)) }
       } ->
       close { send!(fh:handle) }
    }
    Server {
       open  { RPC.Server ((name:string), (fh:handle)) } ->
       loop {
         read  { RPC.Server ((fh:handle, bytes:int),
                             (buf:buffer, bytes:int)) }
       } ->
       close { receive?(fh:handle) }
    }
  }

  Interactions {
    -- Client operations result in corresponding server
    -- operations.  Operations are performed sequentially.
    RPC(Client.open, Server.open) ->
    RPC(Client.read, Server.read) ->
    Client.close -> Server.close {*= prev}
  }
}
```

Notice that iteration is only specified in the role definitions: this minimizes unnecessary specification and avoids the possibility of conflicting iteration constructs in the role and interaction specifications.

2.4.3 Multicast RPC

The original RPC binding can be extended to support multicast RPC. The client and server roles are unmodified, allowing the original client and server to be used:

```
Binding MultiRPC {
  Import RPC;
  Roles {
    Client { RPC.Client }
    -- the cardinality constraint specifies that there
    -- must be at least one server.
    [#>=1] Server { RPC.Server }
  }

  Interactions {
     -- a client send causes all servers to receive
     Client.send -> [#=all] Server.receive {*= prev} AND

     -- however, only one of the responses causes a
     -- result to be delivered to the client.
     [#=1] Server.send -> Client.receive {*= prev}
  }
}
```

This example introduces cardinality constraints associated with roles and their behaviour. All roles in a binding can potentially be filled by many participating components. By default, a role is filled by only one participant. The addition of an appropriate cardinality constraint allows a role to be filled by multiple participants. This use of cardinality constraints provides a convenient and powerful mechanism for describing group communication.

2.4.4 Replicated File Access

A replicated file access binding shows how the multicast RPC binding can be used:

```
Binding ReplFileIO {
  -- replicated, read-only file access

  Import MultiRPC, FileIO;

  Roles {
    -- Client and Servers implement open/read/close
    -- operations, as before.  Only Server cardinality
    -- has changed.
    Client { FileIO.Client }
    [#>=1] Server { FileIO.Server }
  }
```

```
  Interactions {
    -- RPCs by client are multicast to servers
    MultiRPC(Client.open, Server.open) ->
    MultiRPC(Client.read, Server.read) ->
    Client.close -> [#=all] Server.close {*= prev}
  }
}
```

This set of examples demonstrates how a basic interaction mechanism can be extended to suit new requirements. Notice in particular, that clients and servers are unchanged despite the change in interaction mechanism. This suggests significant potential for reuse and legacy application integration.

2.5 Runtime Infrastructure

The *Finesse* runtime infrastructure implements a fully-distributed and asynchronous state machine for executing *Finesse* programs. Each participating component has a local runtime engine that determines when local events can be executed and how they should be positioned in the state machine. Chapter 6 describes this behaviour in more detail. The distribution and asynchrony are possible because the semantic model uses causality as the basis for defining event dependencies: an event can be executed as soon as the local runtime engine has been notified of all events upon which it depends. There is no requirement for the engines to maintain a synchronized view of system state. Each runtime engine thus maintains an incomplete view of the system state, and need only be notified of remote events that are required to satisfy local dependencies.

For example, in the RPC binding described in the preceding section, the client need only be informed of the occurrence of the server send event. While this might seem obvious, it means that the server receive event does not appear (or need to appear) in the client state machine. What is important, however, is that event notifications carry sufficient information to determine where in the state machine they should appear. This is achieved in the runtime engine by using a form of vector clock to capture causal dependencies. Note that the restricted context created by a binding and the finite nature of binding programs means that the vector clocks are bounded in size in

most programs. This aspect is discussed in more detail in chapter 6.

Programs are stored in an internal form as a set of event templates, with decision trees representing the dependencies of each event. An event is executed when sufficient causal predecessors have occurred to satisfy the decision tree and any parameter or timing relationships. The current prototype also requires the participation of the component in all local events, however, this requirement is primarily to simplify implementation and can be removed. When an event is executed, a notification is sent to all remote components potentially having direct causal dependents or parameter relationships.

The key advantage of an asynchronous approach based on causal relationships is that *Finesse* programs can describe and support applications using unreliable or sporadically connected components. This is important in such large scale networks as the Internet and for mobile computing systems.

The runtime engine also supports the time and parameter relationship semantics of the language. While the examples above use name equivalence specification of parameter relationships, these relationships can also explicitly identify only the necessary relationships. This allows the runtime infrastructure to send only the necessary parameters of an event with each notification, which can be a significant optimization when parameters are unused by a remote component.

The prototype currently supports only the Java environment and TCP/IP as the network infrastructure. The *Finesse* language and runtime design are not in any way tied to this platform, however.

2.6 Concluding Remarks

This chapter has introduced the *Finesse* system in an informal manner. The system comprises three key components: an executable semantic model based on the behavioural model informally described here, the *Finesse* language, and a runtime engine. The simple examples presented here hint at the capabilities of the approach. Subsequent chapters will describe these components in more detail and further demonstrate the novelty and strengths of *Finesse*.

Chapter 3

Literature Review

This chapter surveys the literature and technology from research disciplines and commercial products that influence the *Finesse* approach. The goal is to provide a historical and technological basis for the subsequent description of *Finesse* and to provide motivation for the approach and features of *Finesse*.

The chapter first surveys research in software architecture (3.1 and 3.2), coordination languages (3.3), and component models (3.4), which provide much of the architectural basis for the *Finesse* approach. The behavioural model and underling execution semantics is strongly influenced by existing work in distributed systems, specifically interaction models for distributed software components and the underlying middleware technology. These technologies are surveyed in section 3.5. The implementation of *Finesse* is influenced by distributed systems infrastructure technology and this technology is described in section 3.6. Finally, CSCW literature is reviewed in section 3.7, since it is one of the key motivators for this work.

3.1 Software Architecture

A significant influence on *Finesse* has been work in software architecture. Software architecture focuses on the need for concise, well-defined abstractions when constructing software systems. The study of software architecture in recent years has highlighted the need to describe the configuration and interactions of software components, distributed or otherwise, in a way that distinguishes this description from the components

themselves. This separation decouples the behaviour of interacting objects to promote reuse. In distributed systems, this distinct specification of interactions can result in significant optimizations of communication[62]. Garlan and Shaw[118] are pioneers in the area, and their work has resulted in a number of *architecture description languages* [116, 1] and other approaches to capturing software architecture[45]. Typically, architecture description languages allow the description of a set of software component interfaces and the way these interfaces are connected. The semantics of connections are captured in *connectors*.

The Unicon[116] language provides a fixed range of connectors including procedure calls, pipes and filters. In Wright[1], connectors are specified in a language based on CSP[59]. Wright uses modified CSP tools to analyse connectors and the interfaces they connect ensuring that the interactions are, for example, deadlock free. This approach is oriented towards the capture of process-oriented behaviour and uses an interleaved concurrency model. Rapide[80] uses an event based model and posets (partially ordered sets) to capture the true concurrency of distributed systems. Rapide includes a data model and has a toolset that supports simulation and analysis of software architectures. Darwin[84] defines a configuration language that uses process-oriented semantic descriptions of interfaces and connections to determine the correctness of a configuration of objects. This work has a formal semantics based on the π-calculus[94].

These architecture description languages share a common goal of allowing software engineers to accurately document the software architecture. This allows rigorous analysis at the architectural level, reducing the cost of software development by identifying potential problems early in the software lifecycle[118]. These tools are not, in general, augmented by tools to assist in building a software system using the specified architecture.

A number of object-oriented methodologies have also been developed to cater for the needs of object-oriented software developers. Examples are Booch[19] and Rumbaugh[112]. These methodologies have associated languages oriented towards non-distributed software, although work on a methodology known as *UML*[104] is now addressing distributed systems issues. As with the work in architecture description languages, these methodologies recognize the need to explicitly model the rela-

tionships between components (objects), although the focus here is on data rather than process-oriented relationships. UML is making progress towards a formal underpinning, but these object-oriented methodologies lack the formalism necessary to perform rigorous analysis of the described architecture. In contrast to the architecture description languages, however, these methods do have tools and infrastructure to support software development based on the architecture description.

3.2 Distributed Systems Architecture

The need for sound software architecture becomes even more critical when the software is distributed, and recent work on distributed systems architecture was the primary influence in beginning the research described in this thesis. Distributed systems architecture is a relatively new field of research. Literature in the field concentrates on the need to abstract over the implementation detail of a distributed system and capture the essential high-level features. This high-level description should then be mapped onto lower-level software and communication protocols.

Research in distributed systems architecture has grown from the need to step back from the protocol-level approach taken in many early distributed systems. This process of abstraction allows the designer to concentrate on distributed application requirements rather than the nuts-and-bolts of distributed systems construction. It can also enhance portability, with the design not dependent on a particular underlying system.

The ANSA architectural model[86] is widely considered to be the earliest work in this field. It introduced the idea of having five *viewpoints*, namely *Enterprise*, *Information*, *Computational*, *Engineering* and *Technology*. Each viewpoint focused on a specific set of concerns related to a distributed application.

An ISO standardization process for the architecture of Open Distributed Systems was established, and the ANSA model was very influential in the production of this standard. The standard has since been released, and is a four-part standard entitled *The Basic Reference Model of Open Distributed Processing*[63, 105] or RM-ODP. The primary goal of the model is to provide a framework for building distributed programming environments that capture a set of standard architectural principles.

The CRC for Distributed Systems Technology (DSTC) participated in the RM-ODP standardization process and developed the A1$\surd$ Architecture Model[11] to fuel that participation and meet the needs of their organization. The following subsections discuss the A1$\surd$ model, RM-ODP, and related work in more detail.

3.2.1 Reference Model of Open Distributed Processing

Work on RM-ODP[63, 105] began in the late 1980s with the establishment of an ISO/CCITT standards working group. The bulk of the standardization exists in the definition of *computational* and *engineering* viewpoints taken from the ANSA model. The computational viewpoint focuses on the interfaces of objects and interactions between objects. Three types of interfaces are permitted:

1. *operational* interfaces, which exhibit RPC client or server behaviour;
2. *stream* interfaces, which exhibit producer or consumer behaviour;
3. *signal* interfaces, which allow the description of any behaviour, and require an explicit *binding object*;

The behaviour of bindings between operational interfaces and stream interfaces are prescribed by the standard. Signal interfaces only describe local interface behaviour, allowing interaction behaviour to be specified in an explicit binding object. An explicit binding object is equivalent to the connectors used in the software architecture research discussed in the previous section.

The engineering viewpoint describes a model for distributed systems infrastructure, focusing on the creation and maintenance of bindings between interfaces. In effect, this viewpoint is where where the software architecture is mapped onto a distributed systems infrastructure.

3.2.2 The A1$\surd$ Architecture Model

The A1$\surd$ Architecture Model[11] departs from the ANSA viewpoints and focuses on two sub-models: a *specification* model and an *infrastructure* model. The specification model is used to describe the abstract architecture of a distributed system based on the

concepts of *object*, *interface* and *binding*. Objects encapsulate application functionality, interfaces describe the interaction of an object with its environment, and bindings describe the context for interaction between objects. The specification model is similar in many respects to the RM-ODP computational model, but is more flexible and does not prescribe semantics for operational (RPC) or stream behaviour. The explicit use of bindings in the specification model was influential in the addition of signal interfaces to the RM-ODP—early versions of RM-ODP allowed only operational and stream interactions.

The infrastructure model describes a general architecture for distributed systems infrastructure and a mapping between specification model entities and this infrastructure. It is quite similar to the RM-ODP engineering model, but is less prescriptive. The A1√ model does, however, make explicit statements about the relationship between entities in the infrastructure and specification models.

In chapter 4, this thesis refines the specification model to generate a semantic model for describing distributed applications. The semantic model focuses on description of behaviour for bindings and interfaces. The A1√ model has also given rise to other work, including an architecture for resource discovery[73], an architecture for business contracts[95], a type model[20] and an infrastructure that reflects the model[6].

3.3 Coordination

Coordination languages and models complement the recent work in software architecture for distributed systems. This research discipline focuses on the need to program the interaction between software components in a way that distinguishes component behaviour from interaction behaviour. This separation of concerns maps nicely onto the A1√ model concepts of object, interface, and binding.

Research into coordination and coordination languages began with Linda[46]. Linda is a language for parallel and distributed systems based on the notion of a shared *tuple space*. Linda includes a set of basic operations to add to and retrieve from the tuple space. Tuples to be retrieved are selected by regular expressions. The primary

advantages of Linda are the cleanness and simplicity of its model and the decoupling provided by anonymous communication. The primary difficulties are:

1. Any problem that requires an explicit locality (e.g. multiple tuple spaces), real-time constraints, or sorting of tuples is inherently difficult because of the Linda model[5, 21].

2. Efficient implementation over a distributed system is difficult because of the underlying shared memory model which requires reliability[111], and the inherent unreliability of a large network like the Internet.

3. The coordination aspects of an application written using Linda are embedded in the application—there is no explicit representation, hence it is difficult to reason about application interactions without involving the applications themselves[5].

Recent work in coordination languages and systems has moved in a direction more similar to architecture description languages, and has been influenced to some extent by RM-ODP. Relevant examples include:

- ConCoord[60], which provides a flexible language environment for programming both objects and their coordination. Object interfaces are described by ports and states, with an explicit termination state. The coordination language (CCL) describes the connection of ports and the datatypes passed over the connections. It allows the organization of coordinators into hierarchies, providing scalability and abstraction. The language has a fixed set of data types, and appears oriented towards pipe/filter architectures.

- The Coordination Language Facility (CLF)[3], which uses a process-oriented language to coordinate interactions between objects based with CORBA interfaces. The facilities for coordination allow intelligent configuration of objects based on declarative rules. It supports only RPC-style interaction between objects.

- Contracts[58] which is coordination language intended for non-distributed object systems. This language is one of the earliest examples of the separation

between components and their connectors, but does not address distributed systems issues.

- LAURA[71], which is an object-oriented variant of Linda that implements an *offer space* based on the RM-ODP Trader, and supports anonymous operational (RPC) interaction with that offer space.

All of these examples, while advancing research in this arena, are intended for tightly coupled interaction. They are not generally sufficient to support the loosely coupled, dynamic, and unreliable interactions that occur in large-scale distributed systems.

3.4 Component Systems

The notion of "components" has recently emerged as the underlying architectural abstraction for many distributed systems and supporting infrastructures. The definition of this abstraction is widely argued but captured well in [126]. Component systems focus on the need to connect peer components to build large software systems, and typically go hand-in-hand with object-oriented systems. A component is defined by the set of methods it offers to its environment, and the set of methods it expects the environment to offer in return.

There are some significant advantages associated with components. The primary advantage is that a component is self sufficient except for the explicitly defined interactions with its environment. Such a clear definition of dependencies makes components considerably easier to reuse than traditional objects, which tend to have dependencies buried deep inside an inheritance hierarchy. Components also tend to be self describing, allowing the environment to ask the component to describe its interface and hence promote dynamic coupling of components[107].

The self-sufficiency of components reflects the principles of the architecture description languages discussed in section 3.1. The key area of research in component systems, however, is the problem of interconnection. Direct connection of components using local or remote method invocation is provided by most environments[107, 123].

While this is suitable for new, tightly-coupled applications, the connection of existing, loosely-coupled components (i.e. legacy components) requires adaptor objects and such services as "bean boxes"[123]. Adoption of the notion of *connectors* from architecture description languages is not yet a widely supported technique, although some researchers have addressed the issues[58]. Distributed support for connectors is minimal or non-existant.

A further key deficiency of component systems is the lack of facilities for decribing interface semantics above and beyond method signatures. It is not possible, for example, to describe the implicit relationship between data elements in streaming behaviour: only a method to accept a data element can be described.

3.5 Interaction Paradigms for Distributed Systems

Research into distributed systems has been carried out for several decades. Literature in the field encompasses detailed discussion of low-level protocols, the design of infrastructure services to support distributed systems, and appropriate languages and interaction paradigms for programming distributed applications.

A key contribution of this thesis is the emphasis it places on supporting different interaction protocols and paradigms in distributed systems. The following subsections discuss the interaction paradigms and programming abstractions that have emerged from distributed systems research and discusses their relative strengths and weaknesses. The goal is to emphasize that there is no single interaction model or abstraction that satisfies all needs.

3.5.1 Remote Procedure Call

Remote procedure call (RPC)[15] is perhaps the most popular interaction protocol because of its similarity to local procedure call. Use of remote procedure call typically involves the definition of remote procedures using an *interface definition language* (IDL) and the creation of program stubs that approximate local procedure call semantics for the client (caller) and server (receiver). It is therefore relatively easy to modify existing programs for distribution.

Many flavours of RPC exist, each providing different semantic guarantees and data typing. CORBA remote method invocation[98], for example, has a strong object-oriented flavour and supports multiple programming language bindings, where DCE RPC[110] is relatively unique in its support for data pointers. SunRPC[124] is perhaps the most widely used because reference implementations are freely available. SunRPC is the basis for a number of common distributed services, including the Network File System (NFS).

While RPC is generally easy to use, it is not suitable for all applications. In particular:

- RPCs are inherently synchronous, which limits opportunities for parallelism and causes performance problems on networks of high latency.

- RPC implies procedural interaction, so are unsuitable for applications requiring streamed data, for example.

- RPCs are *directed*, in that they require explicit identification of the server by the client. Anonymous or mediated communication gives considerably more flexibility and opportunity for reuse of application components.

3.5.2 Message Passing

Message passing is also quite common in distributed applications and environments. Message passing is used instead of RPC where the interaction model is not strictly synchronous or parallelism in communication is required.

PVM[125] is a commonly used distributed environment based on message passing, with a focus on high-performance parallel programming. Reliable, transactional message passing is provided by environments such as IBM's commercial MQ-series product[50]. At an abstract level, electronic mail (email) is a form of asynchronous message passing, and this has gained wide acceptance in both research and commercial environments.

Message passing can also provide a basis for the streamed communication, although it is more common for stream-oriented communications to be provided explic-

itly by an environment. Message passing has some drawbacks in distributed applications, particularly:

- Message passing is relatively low-level and provides minimal abstraction for higher-level application protocols.
- As with RPC, message passing is directed, requiring explicit identification of the receiver and constraining reuse of application components.

3.5.3 Multicast

Multicast communication protocols have been an area of active research for some time. Multicast typically extends message passing with some notion of distributed, addressable process or object groups. Many protocols and implementations exist, for example in Isis[13], Psync[100], Electra[82], Horus[130] and PVM[125]. The focus of most implementations is on providing reliability through replication. Perhaps the most difficult aspect of providing multicast is choosing an appropriate semantics, that is, how the ordering and delivery of messages is coordinated by receiving objects. It is generally accepted that no single model is appropriate for all applications[14, 12, 27].

Multicast provides a level of indirection over message passing, but most implementations require that participating objects are explicitly aware of the communication model. This typically means that application components are programmed with implicit knowledge of the multicast semantics and can be difficult to reuse in a different communications environment.

3.5.4 Streams

Communicating with data streams has traditionally been the realm of telecommunications providers connecting hardware devices. With the rapid increase in bandwidth available for data communications, audio and video conferencing applications like VAT[64], VIC[90] and NV[43] have appeared, but they have been programmed directly on the transport layer.

A stream abstraction is a necessary component of modern distributed systems[17, 53], and it is clear that quality of service properties must be supported. Few dis-

tributed environments provide explicit support for data streams, although DCE[110] has a functional implementation without quality of service support, and the RM-ODP model explicitly describes stream communication.

3.5.5 Replication

Replication is a commonly used technique in distributed systems. There are many variants, particularly in the way access to replicas is synchronized. Highly-synchronous, single-copy equivalent replication is ideal, but near-impossible to achieve efficiently over an unreliable network. More practical implementations allow some level of divergence between replicas, for example CODA[114], Bayou[128], lazy replication[76] and Prospero[35].

System support for replication usually provides a single semantic model with the better implementations allowing significant configuration. Recent research in CSCW [52] suggests that, as with multicast, no single replication model is appropriate for all applications.

3.5.6 Distributed Shared Memory

Distributed shared memory implementations are an abstraction over replication schemes that allow distributed application components to interact through a synchronous, logically shared, address space. The granularity of access ranges from logical program variables in PARLOG[32] to a shared *tuple space* as used in Linda[46]. These techniques provide a simple and familiar abstraction for programmers, but do not scale well or cope with unreliable networks because of the need for regular synchronization to ensure the consistency of the memory space. More recently, the concept of distributed shared memory has been popularized by JINI[131], yet it retains the inherent limitations described.

3.5.7 Distributed File Systems

While not strictly an interaction paradigm, distributed file systems can provide a convenient mechanism for collaboration. People or applications can share data through access to files that are either replicated, cached locally, or served as required. Distributed

file systems are very similar to distributed shared memory, except that they typically allow some level of divergence to reduce synchronization requirements. Since they use a standard file system interface, distributed file systems can be used by existing applications transparently.

NFS[113] is the most common implementation, and due to its stateless architecture it is highly resilient to failure. It uses minimal caching, however, and hence requires reasonably high bandwidth and cannot support disconnected operation. SMB [92] provides a similar implementation with a focus on PCs rather than Unix workstations. AFS[61], CODA[114] and Ficus[57] are examples supporting replication and disconnected operation. Although it is not strictly a distributed filesystem, the CVS configuration management system[8] uses replication and merging to allow multiple people to work with their own copies of files concurrently.

Distributed file systems are convenient mechanisms for interaction between people, but they do not capture or help manage the complexity of that interaction, nor do they support stream-based interaction. The input/output overhead also tends to be too slow for real-time access to data, thus requiring the addition of caching semantics.

3.5.8 Transactions

Consistency of information is a key requirement for many business systems. In distributed systems, this requirement is typically satisfied by providing a transaction subsystem with some level of guaranteed consistency. In centralized systems, the usual requirement is that transactions satisfy the ACID properties[24]. This can be provided in distributed systems using two-phase commit[51], but the cost in communications bandwidth, efficiency, and latency is relatively high.

A wide variety of methods for weakening the ACID properties in a controlled manner have been described, including several variants of nested transactions[39], lazy replication[76], transactional workflows[48] and transactional messaging[50]. Each have their relative strengths and weaknesses, and are appropriate for different applications.

More recently, research in distributed systems has focused on ways of integrating arbitrary transaction models into a unifying framework, including ACTA[30] and

TSME[47] . These frameworks are typically implemented over an interaction model like message passing or RPC.

3.5.9 Event-based Interaction

The use of *events* for reporting and control is a well-understood technique. More recently, however, researchers and programmers have begun using event-based interaction for programming distributed applications. Objects interact with their environment by producing and consuming events, with produced events routed to the required set of consumers by a mediator.

A common model is the *publish-subscribe* model[75, 87], where consumers select events by *subscribing* to some class or pattern of events via the mediator. Any produced event that matches the pattern is delivered to the consumer. This model of anonymous communication is very powerful, but it is difficult to capture the application architecture and communication patterns from a set of subscription requests.

A more powerful model is implemented in Rapide[80], where the routing of events is explicitly specified by declarative rules. Rules in Rapide are specified as trigger/action pairs, where triggers are event patterns that must be matched. The current implementation of Rapide supports simulation and analysis rather than a runtime infrastructure for programs.

Event-based models are highly flexible and promote reuse because communication between objects is directed by a mediator rather than the objects themselves. This means that participating components are decoupled from the routing of communications, and hence promotes reuse by allowing components to be arbitrarily connected. The inherent asynchrony of event-based systems also makes them more amenable to high-latency, unreliable networks. The primary difficulty with event-based models is that they are less familiar to programmers and hence require a change in design and coding habits.

3.5.10 Connection-based Interaction

Connection-based interaction explicitly connects behaviours specified at the interfaces of components. As with event-based interaction, objects do not name the recipient of

communications, which decouples the objects and hence promotes reuse. The power of this approach depends on the range and flexibility of connectors. A number of systems that implement this approach are described in sections 3.1 and 3.3.

Connection-based models are particularly useful for stream-oriented or pipe/filter architectures. They are more flexible than models based on directed communication, but still suffer from the need for strict compatibility of connected interfaces since communications are not mediated. They are also less abstract than event-based models, providing fewer opportunities for optimization.

3.5.11 Intelligent Agents

Intelligent agent technology[135] provides tools for defining highly adaptable software agents to represent people, organizations, or software components in interactions with external parties. In terms of distributed systems, agents therefore present an external interface for interaction in a distributed systems. The technologies typically include some form of agent communication language or primitives for communication between remote agents. A common language is KQML[88], which is a high-level and quite flexible language for expressing queries between intelligent agents. Communications are strongly directed, and the language relies on statically-defined interaction protocols specified outside the language, with a standard set based on existing interaction protocols. April[89] is an older language that includes both agent and communication primitives, with communication behaviour described in terms of messages, a message buffer, and pattern matching across that buffer.

Intelligent agent systems are typically single-language environments focused on the resolution of complex problems using artificial intelligence techniques. As such, they do not provide a generic platform for distributed systems, but the approaches embody many useful techniques for abstracting and managing communications.

3.5.12 Mobile Agents

Mobile agents build on intelligent agent technology and augment or replace remote agent communication with the ability of an agent to be moved from one platform to another and thus communicate directly with local software systems. Distributed

applications written using a mobile software agent approach rely on creating self-contained, autonomous, mobile software objects that can be passed between cooperating systems[29]. The key abstraction is that both data and code (information and operational semantics) are passed between systems, thus preserving the consistency of the data. This approach is very effective in purpose-written and tightly-coupled systems. It suffers, however, from the inherent problem of requiring significant trust across all participants (you can never guarantee that a participant respect agent "boundaries")[28], and that all systems must provide a consistent environment for execution of the agent.

3.6 Distributed Systems Infrastructure

Distributed systems infrastructure or *middleware* provides the basis for implementing the programming abstractions described in the preceding section. A number of middleware platforms from both research and commercial projects are described in this section. The previous section showed that the set of useful interaction models is quite large and growing. This section highlights the fact that most existing middleware platforms provide only a small set of static interaction models.

Early efforts in distributed systems tended to focus on tightly integrated environments for programming parallel and distributed applications, predominantly distributed operating systems like Ameoba[127] and V [26] and closed language environments like Emerald[16], Argus[79] and Orca[7]. The closed nature of these environments made it difficult to introduce distributed applications into the regular operating environments of computer user, although many significant advances resulted from the research.

The increasing focus on openness in recent years, evidenced by strong support for both de-jure and de-facto standards like CORBA[98], DCE[110] and SunRPC[124], has led to a more open approach that extends existing programming and operating environments. Distributed system infrastructures provide a programming environment and set of services for building and deploying distributed applications. The services and programming environment vary widely between systems, depending on the focus.

This section describes a number of distributed system infrastructures from both

research and commercial organizations. The assertion from section 1.2 of chapter one suggests the need to support flexible interation paradigms, so this assessment has a focus on that requirement. Note the intention of the section is to provide an overview of available technologies, so many existing systems are not described.

3.6.1 Commercial Environments

SunRPC[124] from Sun Microsystems was the first widely used environment for distributed applications, and many applications based on SunRPC are still in widespread use, in particular NFS. As the name suggests, SunRPC is based on the remote procedure call paradigm, and includes an IDL compiler, a data description language (XDR), a rudimentary naming service, and hooks for implementing security. It provides no support for message passing or other interaction protocols, but does allow selection of transport protocols. The RPC semantics depend on the transport chosen. SunRPC is intended for the C programming language only.

DCE[110] from the Open Software Foundation (OSF) attempted to address some of the deficiencies of SunRPC by providing a similar RPC-based system with strong security, a distributed name service, a time service, and support for stream interfaces. DCE was also designed specifically for C language programming. DCE has largely been overtaken by CORBA compliant products and Microsoft's DCOM (a derivative of DCE) in recent years.

CORBA[98] from the Object Management Group (OMG) is a consortium standard defining an interface definition language, remote object invocation semantics (based on RPC), and multiple language bindings to provide language independence. A variety of implementations exist in multiple programming languages including C, C++, Smalltalk and Java. In conjunction with CORBA, OMG is currently defining a number of standards for system services, including naming, event management, security, transactions, and an RM-ODP compliant trading service. CORBA is still limited to an RPC style of interaction, although recent work on event services is addressing this deficiency to some extent.

Lotus Notes[85] is a system that supports distributed applications through a file replication and workflows. The environment provides minimal facilities for building

distributed applications, but is significant because of its wide acceptance and use in commercial organizations.

Tuxedo[2] is one of several transaction-oriented distributed commercial distributed environments. It is primarily concerned with building so-called *3-tier* applications, where a database-oriented client application is shielded from the details of database access by a *middle-tier* that encodes business processes/rules and consistency requirements. Tuxedo provides tools for building the middle tier, including support for distributed transactions using two phase commit, a publish/subscribe mediator, and transactional messaging. Transarc **Encina**[119] provides a similar infrastructure.

3.6.2 Research Prototypes

Amoeba[127] is a distributed operating system from the Vrije Universiteit, Amsterdam. Although not supporting an open systems model, ideas from Amoeba have been very influential in distributed systems research. Its facilities include a distributed programming language Orca[7] based on distributed shared memory, transactional filesystem access using optimistic concurrency control, and high-performance, reliable multicast. Its interaction model at the programming level is predominantly based on RPC. The **V** distributed operating system[26] provided similar facilities.

Isis[13] and **Horus**[130] are research prototypes from Cornell University providing toolkits for distributed systems based on reliable multicast and virtual synchrony[14]. Isis has been developed into a commercial prototype, with Horus being used as the vehicle for more recent research results. Horus provides a selection of possible semantics for multicast, each aimed at different applications. Both systems provide only a group communication abstraction, which limits their usefulness for general distributed systems construction. The ordering guarantees associated with the multicast semantics make these systems excellent for high-availability applications in a local-area network, but can impose severe performance penalties in low-latency, unreliable networks. Similar facilities are also provided by a number of other toolkits, including Electra[82] and it's commercial derivative iBus[83].

ILU (inter-language unification)[66] is a distributed systems toolkit from Xerox Parc aimed at providing language-independent development of distributed applica-

tions. It is based on the RPC interaction paradigm, and is similar in many respects to CORBA. It can interoperate with CORBA applications, but provides stronger compatibility guarantees than CORBA through static checking of strongly typed interfaces.

PVM (parallel virtual machine) provides a library and services for building parallel programs over a network of workstations. It provides message passing and multicast communication, and a number of high-level primitives for managing consistency and supporting parallel applications. Libraries for Fortran, C and C++ are available. PVM is widely used in parallel programming since it is efficient and freely available. The PVM protocols are targetted at high-speed local area networks and as such, it is not generally suitable for high-latency, unreliable networks like the Internet.

Hector[6] is an environment supporting the principles of the A1$\surd$ model. Arbitrary interaction protocols are supported by having component interfaces connected by bindings. Interfaces are implemented as parallel state machines, with binding semantics (interaction protocols) supported by passing messages between the state machines of distinct interfaces. Arbitrary transport protocols can be *plugged in* to support reliability and other constraints. The environment is written in Python[81] and at present requires hand-coding of state machines and interaction protocols.

3.7 Computer Supported Cooperative Work

A final influence on the work reported in this thesis has been recent research into systems supporting cooperative work, otherwise known as CSCW systems. These systems tend to put heavy demands on distributed systems infrastructure, and researchers in this field are openly critical of existing distributed systems, in particular, the lack of flexibility and programmability in the interaction models provided[17]. Their criticisms provide motivation and direction for the thesis.

Research into computer supported cooperative work (CSCW) is carried out by an eclectic mix of researchers from various fields including computer science, sociology, ethnography and education. The primary aim of CSCW applications is to support cooperation between people through computer hardware and software. Most CSCW applications also aim to support physically distant cooperation and hence require a

distributed systems infrastructure. The following subsections discuss CSCW literature relevant to the construction of distributed systems infrastructure for CSCW applications.

3.7.1 CSCW Systems and Sociological Theory

There has been a significant evolution of CSCW systems in the last five years. Early systems attempted to formalize work practices in static, rigid environments. To a large extent, these systems were unsuccessful because of their inflexibility. CSCW researchers looked for theories and models that could guide the construction of more flexible and workable systems. Speech act theory[133] was embodied in a number of implementations, for example, Conversation Builder[69] and ActionWorkFlow[91]. While more flexible than the previous attempts, these were still relatively unusable because they relied on the ability to formalize work processes.

The sociological theory of Strauss[122] suggests that work processes are inherently driven by continuously evolving social structures. Contingencies are also the rule rather than the exception, suggesting CSCW systems that do not support continuous evolution of the work process are unlikely to be successful. The theories of Strauss are supported by experience with CSCW systems[67].

CSCW systems are now beginning to take these issues into account, and focus on providing tools to facilitate interaction rather than tools that constrain interaction, or in other words, providing tailorable mechanisms not policy. Approaches based on interaction through shared workspaces like wOrlds[42], DIVA[120], POLITeam[72], TeamRooms[109] and MASSIVE[53] support these principles. A key issue in these systems has been the implementation of awareness mechanisms, allowing participants to be aware of each other's actions.

A weakness in shared-workspace models is that they constrain interaction to a virtual space and do not adequately capture the social aspects of work. Recent work reported in [41] and [56] suggests that these spatial models need to be replaced or overlaid with models that capture the overlapping social groupings that exist independent of location. Orbit[68] is a research prototype that provides non-spatial mechanisms allowing users to participate in many social groups concurrently, but continuing to use

a spatial techniques to group related resources.

3.7.2 CSCW Toolkits

CSCW toolkits provide libraries and tools that implement programming abstractions suitable for constructing CSCW applications. These toolkits are often equivalent to distributed systems infrastructures, but with a focus on interaction mechanisms needed for CSCW applications including multicast, replication, floor control mechanisms, and event-based awareness mechanisms. GroupKit[108], Intermezzo[38], COAST[115] and Habanero[96] are examples. The fact that these toolkits exist suggests that existing distributed systems infrastructures do not provide appropriate support for CSCW applications. Even custom-built CSCW toolkits suffer, however, from rigidness of implementation that makes it difficult to customize them to support the requirements of varying applications and work practices[36].

3.7.3 CSCW Criticism of Existing Distributed Systems

CSCW researchers are openly critical of the support for CSCW provided by existing distributed systems. Paul Dourish in his PhD thesis[36] promotes the principle of Open Implementation[70] for CSCW systems. This principle suggests that toolkit implementations and infrastructures should be accessible for configuration and modification through a meta-object protocol. Dourish notes that although he could exert considerable control over local object access in the CLOS language environment, he had no control whatsoever over the communications infrastructure used to interact with remote objects.

Similarly, in a critique of the wOrlds prototype[67], it was noted that although CORBA and related OMG technologies provided a seamless distributed programming environment, they have a fixed interaction model that did not scale or cope with unreliable communications.

Greenhalgh and Benford[53] in their implementation of aura management in MASSIVE found that the directed and synchronous nature of RPC was inappropriate for many aspects of their application. As a result, they suggest that distributed systems need a “richer communications model” to adequately support CSCW applications.

Blair and Rodden[17], in their discussion of the use of RM-ODP for CSCW systems, point out substantial deficiencies in that model, particularly in support for group work in the areas of transactions, security and management. They also highlight the need to avoid “overly prescriptive” viewpoint language specifications, and the need to support multimedia, multiparty communications.

These and other experiences make it clear that a key requirement for distributed systems infrastructure in CSCW systems is to provide flexibility and openness in defining communication and interaction models.

3.7.4 Languages for Describing Collaboration

To overcome the lack of flexibility in infrastructure implementations, a number of systems have experimented with the idea of programming collaboration mechanisms. These systems are quite similar to the coordination languages described in section3.3.

DWCPL[33] is a custom-built language for building synchronous collaborative applications by programming the interactions between objects. The language provides a relatively high-level abstraction of collaboration, and has good facilities for structuring and reuse. Its usefulness is limited by its focus on synchronous applications, thus making it difficult to use for asynchronous group work.

Trellis[44] is a language based on petri-net semantics and is used to describe collaboration protocols. Its primary aim is to support flexible collaboration protocols with a graphically oriented tool, allowing modification by users. The implementation is potentially useful for simple workflows and collaboration protocols, but suffers from a number of limitations, including a static set of states and a lack of facilities for structuring and abstraction.

Introspect[129] is a process specification and execution environment based on Smalltalk and wOrlds[42]. It uses a graphical notation based on Harel statecharts[55] to describe processes for collaboration. The notation includes hierarchical structuring and a reflective architecture allowing on-the-fly modification of behaviour. It also attempts to capture the Strauss[122] notion of *trajectories* that span workspaces and social groups. The system uses a centralized server architecture, however, making it difficult to operate in an unreliable network.

3.8 A Scaffolding supporting Finesse

This chapter has described research and technology in distributed systems to provide a technological basis and motivation for the *Finesse* approach. From this discussion, we suggest that the surveyed technologies do not address the assertions specified in chapter 1. In particular:

- Coordination languages, while supporting the necessary flexibility in component relationships, are not able to deal with high latency and network failure, thus the assertion of 1.3 is not satisfied.
- Existing distributed infrastructures can deal with latency and network failure, but fail to provide the necessary flexibility of interaction or the ability to capture interaction paradigms at a sufficiently high level of abstraction, thus the assertion of 1.2 is not satisfied.
- CSCW applications and toolkits are typically built on the platforms above and inherit their restrictions.

The work on software architecture and architecture description languages suggests the path forward: flexible definition and programming of interaction semantics for distributed components. Distributed component systems go some way towards realising the goal through the explicit capture of dependencies in interfaces. The following chapters of this thesis describe how the *Finesse* system extends these approaches to satisfy the key assertions of chapter 1

Chapter 4

Semantics of Behavioural Model

Chapter 2 gave a high-level overview of the Finesse system. In this chapter[1], the executable semantic model associated with binding behaviour is specified, first informally and then formally using the Z specification language. The focus in this chapter is on the model of behaviour associated with the Finesse system, that is, realising causal, parameter and timing relationships between events. The binding/interface/role structuring described in the chapter 2 is of minimal significance, and the semantic model described here could quite conceivably be used within a different architectural context.

The semantic model presented has emerged from the examination of distributed systems architecture and the realization that static, end-to-end models of interaction like remote procedure call make it difficult to interconnect systems. Early influences include the ISO Basic Reference Model of Open Distributed Processing [63] and the A1$\surd$ model [11, 102], in particular the the notion of a "binding object". The Finesse language introduced in chapter 2 was devised to program binding objects, and during this effort it was realized that a distributed, asynchronous execution model was both possible and desirable. Investigation of similar work in coordination languages showed that most execution models were based either on relatively static connections [4, 60] or required a shared global state abstraction, for example systems based on the Linda tuple-space model [23, 22, 34]. Architecture description languages provided some insights, in particular Rapide [80] and Wright [1], yet had no concrete distributed

[1]This chapter is an extended version of a paper published at the Fifth International Symposium on Autonomous Decentralized Systems[10]

execution semantics. Simulations of architectures are certainly possible, but again rely on a centralized notion of state.

The key distinguishing feature of the model is in describing a distributed, asynchronous execution semantics for binding behaviour. The underlying behavioural model bears a strong resemblance to event structures [134], but no existing work addresses the execution of these semantics in an asynchronous distributed environment. It could also be argued that Petri nets[99] provide an equivalent execution model. While a Petri net specification can be executed in a distributed manner, the Petri net abstraction imposes constraints forcing the co-location or synchronization of certain subsets of behaviour. In addition, the mapping between a Petri net abstraction and typical programming constructs is non-trivial. This is evidenced by the fact that no widely known coordination language or architecture description language uses a Petri net semantic model. The model presented in this chapter is more approachable and avoids these constraints.

In describing the Finesse semantic model, we begin by describing a model for concurrent programs then apply it, with some modification, to distributed programs. The key feature of the model is that with minor constraints, distributed participants can proceed entirely asynchronously except where synchronization is explicitly specified by the program. In other words, the execution model provides a semantic basis for an asynchronous distributed state machine. The following sections informally describe this model in more detail, discussing its properties, strengths, and weaknesses. A more formal description is given beginning in section 4.8.

4.1 Base Execution Model

The semantic model is based on the execution of event templates to create events. Event templates describe causal, parameter and timing relationships between events. An execution of a program executes these templates in a manner consistent with the relationships. The resulting program execution is represented by a graph with each node in the graph representing an event, and each arc representing a causal dependency and hence ordering relationship between the source and destination events. Concurrency is

captured by branching in the graph. A simple program execution is depicted in figure 4.1. As suggested previously, this model of concurrent computation is closely related to that modeled by event structures[134].

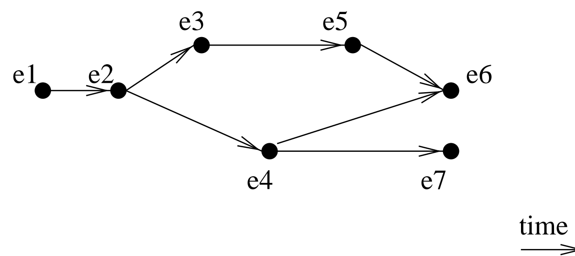


Figure 4.1: Event Execution Graph

The graph in figure 4.1 represents a historical view of an execution, that is, it represents an execution that has occurred. Each event in the graph describes a single occurrence or action. It has a set of causal predecessors which are the sources of incoming arcs in the execution graph, and a set of causal successors, which are the targets of outgoing arcs in the execution graph.

An event also has a set of attributes. These attributes represent information associated with the occurrence of the event, for example, the time at which it occurred, or programmatic data generated by its occurrence. The event attributes and its causal predecessors are immutable: they cannot be changed once the event has occurred. The combination of event ordering and attributes is used to describe a program execution.

The preceding model is only able to describe the completed execution of a program. To enable the description of executable programs using this base model, we introduce a notion of event templates and a history. Event templates define the possible future events than can occur, and the history defines the graph of events that have occurred. Figure 4.2 depicts this model, with the dotted lines between events in the history and templates representing *offers* of dependency from events to templates. These offers can be understood as program sequencing instructions and will be explained in more detail in subsequent sections.

A program executes by choosing templates that are enabled by the history and executing them. The resulting events are added to the history with incoming arcs

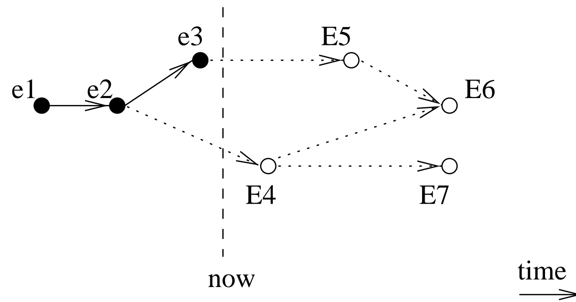


Figure 4.2: Execution State with Templates

reflecting on the offers chosen when the template is executed. For example, if template E_4 from figure 4.2 is executed, the new state of the execution is shown in figure 4.3.

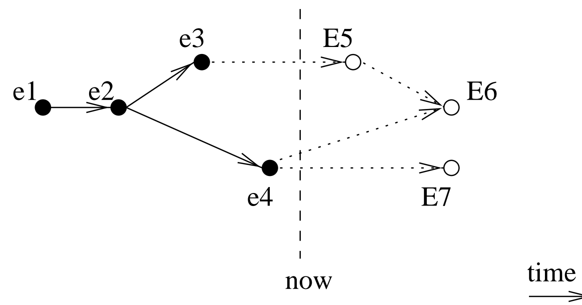


Figure 4.3: Event E_4 Executed from Template

4.2 Templates describe Programs

Templates allow us to describe programs by capturing both ordering and attribute constraints for the events that constitute a program execution, without being tied to a particular occurrence. If we allow for conditional execution of templates, a template must also capture the guards representing program conditions. A program is thus a set of templates. In chapter 5, a language is presented that maps language primitives to a set of such templates.

A program executes by instantiating an empty history, then executing those event templates that are initially enabled, which enables further templates, and so on. Iteration is possible through the re-enablement of a template by subsequent template

execution. In a program execution, a template can be executed when a set of causal predecessors satisfies its incoming ordering constraints and any guards. The template must therefore contain a description of the necessary causal predecessors and guard conditions.

In order for a program to execute an event, it must also assign attributes to the event. The template can constrain the values of those attributes. Typically, such constraints will allow the program to deterministically assign a set of values to event attributes, or in other words, the template will specify values for attributes. These attribute specifications can include references to attributes of causal predecessors, thus decribing the flow of data between events.

It is important to note that template specifications must be independent of specific events. Guards and attribute constraints in a template can therefore only refer to templates. To map these template references to event instances and their attribute values, events must capture their association with a template, and the guards and attribute constraints on a template must be satisfied by the events offering their causality to the template at execution time.

In a sequential program, this would be sufficient information for program execution because events execute in a total order. The presence of concurrency requires that we allow the execution to split into concurrent execution threads. We describe this by having each template describe the finite set of possible, immediate causal successors. When the event is executed, these become the unterminated arcs and can be thought of as *offers* from the event to future events. The offers can be used to satisfy the causal predecessor relationships of templates and thus allow the execution of those templates. The set of causal predecessors and successors associated with a template can involve choices and be non-deterministic, for example, logical AND, OR and XOR relationships between the possible predecessors and successors. This can result in mutually exclusive predecessor relationships, and most importantly, mutually exclusive offers to successors. Figure 4.4 illustrates the logical conditions that might exist across causal successors or predecessors. In this case, both $e3$ and $e4$ must follow $e2$, only one of $e6$ and $e7$ can causally succeed $e4$, and $e6$ can be enabled by either or both of $e5$ and $e4$.

The combined specification of causal predecessors, causal successors, guards and

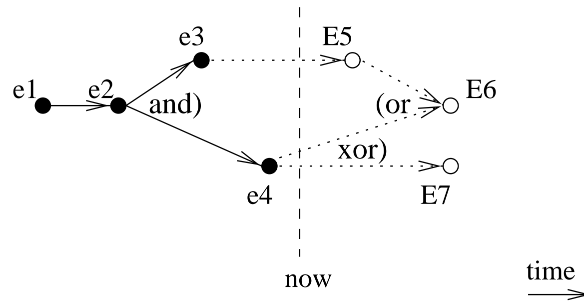


Figure 4.4: Non-deterministic execution

attribute constraints fully describes a template. A program is described by a set of templates where all contained references to other templates are satisfied by the set. The causal predecessors and causal successors capture causal relationships between events, while the guards and attribute constraints capture the timing and parameter relationships.

4.3 Program Execution

As previously suggested, a program execution begins by instantiating an empty history and executing any templates that are initially enabled. These enable subsequent template executions and so on. A template can be executed if:

1. it has sufficient offers from events in the history to satisfy its causal predecessor set;

2. the template guard is satisfied, knowing the choice of predecessors; and

3. the execution of the template will not invalidate the causal successor constraints of any *offers* that are used.

When executing the template, the execution engine must add an event to the history with appropriate arcs to event predecessors and future templates, set the value of the event attributes using the constraints associated with the template and any other influences (e.g. results of local computations), and update the offers of predecessor events in the history to reflect the acceptance of an offer by this event. A key issue is

that when executing events concurrently, care must be taken to ensure that mutually exclusive offers made by an event are not accepted concurrently.

The usual properties of concurrent execution are relatively straightforward to capture in this model, specifically:

- A program can *terminate* when the set of terminated outgoing arcs for each event (accepted offers) satisfies the causal successors constraint of the associated event template.

- A program is *deadlocked* when no template can be executed now or in the future, but the program cannot terminate.

- A program is *livelocked* when one or more templates whose execution is necessary for termination cannot be executed now or in the future, yet other infinitely executable behaviour is possible.

4.4 Parameters

The execution model has thus far focused on the basic control flow associated with program execution. With control flow appropriately defined, we can now address the issue of data flow. Data flow in the semantic model is specified by defining the relationships between event parameters. The value of an event parameter is set by evaluating such relationships or assigning a literal or local, environment-supplied value. To simplify definition of these relationships and make them feasible, we require that any events referenced in a parameter relationship be causal predecessors, or in other words, parameter relationships are defined at the successor event and can only refer to events connected to the successor by the causality graph. This is intuitively correct, since a parameter relationship implies a causal relationship.

4.4.1 Defining Parameter Relationships

The general form of a parameter relationship is $X \in F(Q)$, that is, the parameter X must be assigned a value from the set returned by a function F applied to a set of values Q. The values can be expressed as references to parameters of causally preceding events,

literal values, or environment supplied values. While we allow non-determinism in the value of X by choosing from the set of values returned by F, most implementations (including the one described in the following chapter) would most likely insist that the function F return a single value. There are number of advantages in this approach that should be highlighted:

- A program explicitly captures data dependencies in a declarative manner. The transfer of data from one location to another can thus be optimized. In particular, it is not necessary for all parameters of an event to be transmitted when notification of the event is transmitted. Referring back to figure 4.5, event $e2$ might have parameters (x, y), but if $e3$ refers only to y, only y must be transmitted when notifying $site1$ of event $e2$. It is also interesting to note that a value expression can be evaluated at any location, thus allowing a runtime engine or compiler to determine the most efficient place to perform the evaluation.

- Data mismatches between event attributes can be handled without the involvement of components connected by a program. For example, a request event that outputs a date as a string can be delivered to a receiving component that expresses a date as a number of seconds since an epoch, provided an appropriate mapping function is defined.

- Where the control flow indicates that parameters of multiple events from a single location are required to evaluate a relationship expression, the event notification can be delayed until all required local events have been executed and the notifications combined.

4.4.2 Identifying Event Parameters

A key difficulty in implementing event parameter relationships is identifying the events whose parameters are to be used. A program is expressed in terms of templates, and event references can only use template names to identify other events. Given the presence of iteration and dynamic renaming of templates in the control flow semantics, rules for mapping template names to event instances must be implemented. Since these rules do not change the model of parameter relationships presented above they

can be implementation-dependent. The following rules, however, are a useful starting point.

1. In the presence of iteration, the most recent causal predecessor matching the specified event template name should always be used.

2. Where a role is taken by multiple interfaces (with the implied dynamic renaming of templates), an implementation should either provide a default rule to choose a single event from one of the interfaces (e.g. earliest), or allow all event references to indicate a set of events, and provide functions for manipulating event sets. Note that this rule can be implemented at a language level provided the underlying implementation supports references to event sets.

4.5 Guards and Timing Constraints

Guards in the semantic model are arbitrary boolean expressions that must evaluate to TRUE for an event to be executed. The general form of a guard is $G(Q)$ where G is a boolean valued expression over a set of values Q. The values in Q are specified in the same manner as those used in parameter relationships, that is, they can be references to parameters of causally preceding events, literal values, or local, environment-supplied values. Time is considered one of the local environment-supplied values.

Each event in the history has a timestamp attribute that identifies the time at which the event occurred. The semantic model also requires that the local environment can supply the current time on request, and that time always moves forward. These times, however, are relative only to a reference point determined at program instantiation time. Guards can include expressions that constrain the elapsed time since a previous event or a well-known epoch, but constraints on absolute time cannot be used in guards. For guard evaluation, the semantics requires that the current time since the agreed epoch is an environment supplied value.

The use of time and event references in guards has some interesting consequences, particularly:

- Guards have a time dimension, meaning that a guard that is FALSE at a certain time can subsequently become TRUE due to the passage of time only;

- Guards are unresolvable when contained event references are unresolvable unless the guard can be resolved without evaluating the terms including those references (i.e. short-circuit evaluation). For simplicity, we consider an unresolvable guard to be FALSE.

- An event template can become *impossible* (i.e. can never be executed) if a maximum time limit expires (subject to short-circuit evaluation) or a necessary predecessor withdraws its offer of causality.

Implementations must take these properties into account, in particular, an implementation might need to maintain timers to manage guards involving time constraints. Although absolute time is not used, the semantic model assumes a global clock for time comparisons. In practice, this means that an implementation must take steps to ensure time synchronization and take synchronization accuracy into account when evaluating time constraints. No particular representation of time is mandated, except that a literal value representation of time intervals and a function to return the time interval between two time values must be supported.

4.6 Distributing the Execution

The execution model described in the preceding section does not offer any significant enhancements in expressiveness over other programming and execution models based on true concurrency, for example, Petri nets [99]. The unique feature of the model, however, is that with some constraints, programs expressed in this manner can be executed by a set of distributed state machines with *no synchronization except that explicitly required by the application-level relationships specified in the program.* An implementation of such a distributed state machine is described in chapter 6. While it is possible to distribute the behaviour described by a Petri net, the partitioning of behaviour must respect the synchronization constraints imposed by the Petri net model. No such constraints are imposed by this model.

The execution of a program in the model described in the preceding sections involves modifying and reasoning about an execution history. If we were to require all participants in a distributed execution of the program to operate on a single, consis-

tent view of the history, program execution would be limited by the communication required to synchronize the history each time an event is executed. Our goal is to avoid this synchronization since it is impractical for Internet-scale applications that must tolerate network failure and potentially high latencies in communication. We must therefore allow participants to operate on a local and possibly incomplete view of the execution history.

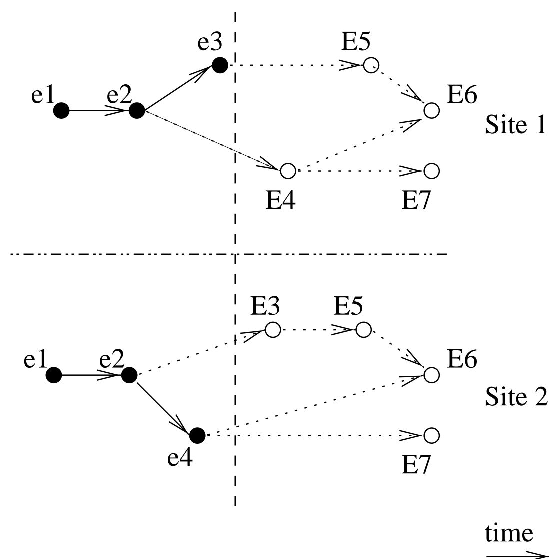


Figure 4.5: Distributed Execution State

Consider the example depicted in figure 4.5. In this example, *Site*1 is aware of the execution of events e_1, e_2 and e_3, while *Site*2 is aware of e_1, e_2, and e_4. The template E_5 is apparently enabled at *Site*1 because its only causal predecessor (e_3) is known to have been executed. Similarly, template E_7 is apparently enabled at *Site*2. In the following discussion, we define the necessary conditions for these templates to be executed against an incomplete view of state.

We begin by examining the requirements for event execution specified in section 4.3. The first requirement was that a template have *sufficient offers from events in the history to satisfy its causal predecessor set*. In other words, the local view of history must contain a set of events that offer their causality to a template and satisfy its causal predecessor set. While notification of remote events might be required, this notification can be asychronous. Once such notifications have occurred, the rule can be correctly

evaluated against a local view of the history.

The second requirement is that *the guard is satisfied, knowing the choice of predecessors*. This can be evaluated against a local view of the history provided that view includes all events referenced by the guard. Similarly, references to causally preceding events in the parameter relationship specification must be satisfied by the history. We address this problem by adding a requirement that an event reference implies that the event referenced is a mandatory member of the causal predecessor set for this event, and that the event referenced must make an offer of causality to this event. Note that the runtime system described subsequently in chapter 6 treats these offers somewhat differently to explicit causality relationships because it maintains a more complete causal history than strictly required.

The third requirement is that *the execution of the template will not invalidate the causal successor constraints of any offers that are used.* This requirement is difficult to ensure, since it implies that we must prevent concurrent acceptance of mutually exclusive offers made by an event. For example, the program depicted in figure 4.4 allowed only one of $e6$ or $e7$ to succeed $e4$, some form of distributed decision must be made across the sites to choose the successor. There are a number of possible solutions:

1. Force synchronization to choose one of the mutually exclusive potential successors when such conditions exist;

2. Nominate a single participant to deterministically choose a correct set of successors at run-time (autocratic choice);

3. Require that non-determinism can be locally resolved by mutually exclusive guards or runtime checks on templates wherever conflicts exist; or

4. Allow optimistic execution and raise an exception when inconsistent behaviour is detected.

This is the key difficulty in distributing a concurrent program, and the following subsections discuss the possibilities in more detail. In summary, however, we conclude that the first and second options above require significant synchronization and

are thus excluded. The preferred solution is to require that non-determinism be locally resolved (3), but programs that cannot satisfy this requirement can be executed using the optimistic method. A program that permits local resolution of non-determinism is called a *safe* program.

Note that the chosen approach moves the problem to the programming language level, meaning that the application programmer or language tools can resolve the non-determinism in a manner that is most efficient and appropriate for the application.

4.6.1 Synchronization

Synchronization can be used to allow choice of non-deterministic successors. The choice could be implemented, for example, through a voting algorithm. While such algorithms are refined and well-understood, a number of round-trip messages must be exchanged by participants. In an unreliable, high-latency environment like the Internet, this can impose a significant overhead so this solution is not considered feasible.

4.6.2 Autocratic Choice

Nominating a single participant to make an autocratic choice of the set of successors is possible, but requires reliable communication to ensure that the chosen successors are explicitly aware of the predecessor. It also requires complete knowledge of the execution context of the chosen successors to ensure that any guards are satisfied and thus avoid deadlock. In other words, the chooser must have knowledge of all predecessor events for the chosen successors to ensure that their guards are satisfied. In the worst case, these requirements mean that the chooser must have a complete view of the history. Since our goal is to avoid the need for such a view, this solution is not considered feasible.

4.6.3 Safe Programs

We use the adjective *safe* to describe programs where all non-determinism in offers made by events can be locally resolved. The following conditions in a program can resolve the non-determinism:

- Conflicting successors of an event have mutually exclusive guards; or
- Conflicting successors of an event execute locally, thus allowing the local state machine to safely choose a single successor; or
- Conflicting successors of an event have other mutually exclusive predecessors, and any non-determinism across those predecessors can be locally resolved.

This problem is one of resolving distributed predicates using local knowledge. A deeper analysis of distributed predicates and their resolution is given by Charron-Bost et al in [25]. In our case, the key problem in ensuring that a program is *safe* is that guards can refer to the parameter values of preceding events that are bound at runtime, and hence mutual exclusion is difficult to recognize in a static parse of a program. It is important, however, to reflect on the types of program constructs that can lead to mutually exclusive offers. Such constraints are imposed by exclusive OR relationships across successors, typically found in programs as *if-then-else* constructs or *case* blocks. It is usual for these constructs to include or imply mutually exclusive guards. In figure 4.4, for example, a choice between the execution of $E6$ and $E7$ must therefore be protected by an *if-then-else* construct where the guard expression can be resolved independently by both sites. In the general case, distributed exclusive OR relationships without mutually exclusive guards imply non-deterministic language semantics (i.e. the system must make a choice). No commonly-used programming language has constructs that allow such non-determinism.

We conclude that while the model can describe behaviour that cannot safely be distributed amongst asynchronous distributed state machines, it is unlikely that programming languages would allow the expression of such behaviour. Languages that can express such behaviour must either do so in a controlled manner or be considered *unsafe*. As discussed in the introductory comments, deferring the problem to the language level in this manner is the preferred solution.

4.6.4 Optimistic Execution

As briefly described above, we can allow asynchronous execution by ignoring potentially inconsistent behaviour and raising an exception if such behaviour is detected.

While this might seem computationally unsound, static analysis of programs can flag potential non-determinism and give the programmer an opportunity to deal with it. In particular:

- the programmer can selectively remove non-determinism through guards or more explicit programming;
- the use of exceptions could allow programmer-specified recovery when inconsistent behaviour is detected, and this is often preferable to synchronization;
- there are a number of situations where non-determinism will be resolved at run-time through notifications of event execution. For example, where two successors have additional, co-located enabling events that are mutually exclusive.

The key advantage of an optimistic approach is that it gives control to the programmer so that possibly inconsistent behaviour can be selectively allowed if the programmer deems it benign or sufficiently unlikely. This is also a disadvantage, since with control comes the need to understand the source of the non-determinism and the potential problems of allowing inconsistent behaviour. A further disadvantage is the fact that it will not always be possible to detect incorrect execution at run-time. Inconsistent behaviour can only be detected when knowledge of the execution of conflicting events is available in one location. Given the truly concurrent nature of the execution, this cannot be guaranteed.

In subsequent discussion of the semantic model, we assume that mutually exclusive offers can be locally resolved, or in other words, only *safe* behaviour is possible. It is expected that implementations of the semantic model will use the optimistic approach and rely on programming languages and tools to avoid *unsafe* behaviour. The execution can therefore proceed entirely on a local view of execution history.

4.7 The Significance of Location

The model described above does not associate event templates with any particular site or location. If a single event template can be executed at multiple locations, we must either allow an offer to be accepted by multiple executions of a template, or explicitly

associate a template with a location. The semantics of allowing multiple acceptances of an offer is neither intuitive nor easy to implement and can lead to unsafe behaviour similar to that discussed in the previous section. We therefore require the program execution to associate an explicit location with each template.

Assigning a single location to each template might sound highly restrictive, but in practice, it reflects common programming models. Consider the following:

- The association of a template with a location can and usually will happen at program instantiation;
- The concepts of *role* and *interface* introduced in chapter 2 give flexible co-location constraints for event templates in a program definition. At program instantiation time, a program role can be associated with each component interface, and the set of templates in the role behaviour can be renamed or annotated with the interface identifier to distinguish their location. Note that this method also works when multiple interfaces implement the same role (e.g. for replication or process groups).

Through ensuring that templates identifiers include location, we can guarantee that an enabled template can only be executed at one location and thus avoid additional non-determinism.

4.8 Formal Specification

The preceding sections have described the *Finesse* semantic model informally, with the key area of distributed control flow explored in some detail. The following sections formalize this model using the Z specification language. The focus in these sections is on control flow, since the parameter and guard semantics are defined in a rigorous fashion in previous sections.

Z is used in a relatively simple and axiomatic fashion that should be accessible to any reader with a basic understanding of logic and discrete mathematics. Z was chosen for two reasons: firstly because axiomatic Z specifications have no underlying execution semantics and we thus avoid any implications of such semantics; and secondly

because of familiarity with the language on the part of the author. To assist in reading the specification, this section presents a brief introduction to the Z specification language. Further detail can be found in [121].

4.9 Introducing Z

Z is a set-based specification language, with the majority of its operators and constructs used to describe sets and relations. Types in Z are sets and are typically defined either as base types or schemas. A base type defines a set of raw elements with no defined subordinate structure, and is named by convention with an upper case word and introduced by placing it in square braces, for example $[MYTYPE]$. Schemas define types of entities that have multiple parts, use capitalized words as names, and are introduced using the following notation:

MySchema

$name : NAME$
$age : \mathbb{N}$

$\forall m_1, m_2 : MySchema \bullet m_1.name = m_2.name \Rightarrow m_1 = m_2$

In other words, the type MySchema has two attributes: *name* from the set *NAME* and *age*, a natural number (indicated by the predefined type $\mathbb{N}$). The portion below the line splitting the box constrains or defines the members of the set, and in this case specifies that members of the set *MySchema* are uniquely named. $LHS \Rightarrow RHS$ in the constraint can be read as "if LHS then RHS". Another similar and often used logical operator is $LHS \Leftrightarrow RHS$ which can be read as "LHS if and only if RHS".

Z also makes extensive use of binary relations. The type of a relation between the sets X and Y is specified as $X \times Y$ and means the set of all possible pairings of elements from the sets X and Y. The elements of the set X are known as the domain of the relation, specified as $\operatorname{dom} R$, and the elements of Y are known as the range and specified as $\operatorname{ran} R$. Infix notation can be used to specify membership of a relation, for example, $x \mathrel{\mathrm{R}} y$ says that the pair (x, y) is in the relation R. Z also provides domain and range restriction operators to extract subsets from a relation. $\{x_1, x_2, .., x_n\} \lhd R$ specifies the set of pairs from the relation R whose domain element is in the set $\{x_1, x_2, .., x_n\}$.

Range restriction is similar, for example $R \rhd \{x_1, x_2, .., x_n\}$ restricts R to the set of pairs whose range element is in $\{x_1, x_2, .., x_n\}$.

A commonly used restricted relation is a function, specified as $X \rightarrow Y$, which is a relation where each element of the set X is paired with exactly one element of the set Y. Z also defines a partial function $X \nrightarrow Y$, where the function is only defined for some subset of X (i.e. the function domain is a subset of X).

Z supports the definition of axioms. In the specification of the semantic model, axioms are typically used to define functions or relations with constraints, for example:

$$
\begin{array}{|l}
\mathit{Identity} : X \rightarrow X \\
\hline
\forall (x_1 \mapsto x_2) : \mathit{Identity} \bullet x_1 = x_2
\end{array}
$$

This defines an identity function over the type X. The upper portion defines the name and type of the axiom, with the lower portion defining the constraints. Note that the $\mapsto$ operator is used to define pairings of elements in a relation.

Z defines a number of common types. The set of natural numbers is denoted $\mathbb{N}$ and the set of real numbers (not formally part of Z but commonly used) is denoted $\mathbb{R}$. Z also defines sequences of arbitrary types, denoted $\mathrm{seq}\, X$. Formally, this defines a relation between natural numbers and the elements of the set X, that is, $\{1 \mapsto x_1, 2 \mapsto x_2, .., n \mapsto x_n\}$ where $\{x_1, x_2, .., x_3\} \subseteq X$. The *head*, *tail* and *last* operators are also defined for sequences.

A final important aspect of type specification in Z is the powerset, denoted $\mathbb{P}\, \mathit{SOMETYPE}$. This defines the set of all possible subsets of the set *SOMETYPE*. When an attribute is given the type $\mathbb{P}\, \mathit{SOMETYPE}$, this says that the attribute is a set of elements from *SOMETYPE* (i.e. a subset of SOMETYPE). The usual mathematical symbols for set and logical composition operations are used in Z.

4.10 Basic Behaviour Description

Causal dependencies between events form the basis of the semantic model, thus we first define a model for execution of events in a program based purely on these causal dependencies. As previously described, a behaviour is a directed, acyclic graph of

events. The execution of a program must therefore result in such a graph. We base the execution semantics on the notion of an initially empty history, which defines a graph of events that have occurred, a set of event templates that define possible future events and their relationships, and a transition function that relates a history and set of templates to a new history containing one or more additional events. A program specification is thus a set of event templates, and an execution "unrolls" that specification to generate a behaviour. Note that iterative behaviour is captured by allowing multiple executions of a template, thus a program need not contain distinct templates for every possible event. The formal execution model will define the notions of event, event graph (i.e. history), event template, the transition function, and correct execution.

4.10.1 Base Types

We first introduce the types $[\mathit{EVENT}]$, $[\mathit{TEMPLATE}]$, $[\mathit{GUARD}]$ and $[\mathit{PARAMREL}]$. These can be interpreted as sets of identifiers for schemas that define the attributes of events, event templates, guards and parameter relationships respectively. We use this approach both to defer specification of some entities, and because Z does not allow recursive schemas thus requiring reference types. Note that Z is case sensitive, so we use the same name to refer to the related schema types, but with only the first letter capitalized.

Events have parameters, but for the purposes of the semantic model, the structure of such parameters is unimportant and it is sufficient to simply assert the existence of a set of parameter values $[\mathit{VALUE}]$, and a set of names $[\mathit{NAME}]$ used to distinguish the parameters of an event.

Time semantics in *Finesse* are relative, that is, it is not necessary to support absolute time. We therefore represent time values as real numbers indicating a number of seconds since an agreed reference point in time. For readability, we define the alias $\mathit{TIME} == \mathbb{R}$. Note that we do not distinguish time by location, or in other words, we assume the existence of clock synchronization across physical locations. In an implementation it will be necessary to consider clock skew when evaluating time semantics. Where absolute time is necessary, applications must introduce an explicit clock.

The application structure in the *Finesse* language are not part of the semantic

model, however, we assert that all modelled behaviour occurs at the interfaces of components and we capture the notion of location by introducing a type $[LOCATION]$.

4.10.2 Events and Event Templates

An *Event* is an immutable entity with a set of parameters, a time, and an event template from which it was executed:

$$
\begin{array}{l}
\textit{Event} \\
\hline
event : EVENT \\
params : NAME \nrightarrow VALUE \\
time : TIME \\
template : TEMPLATE \\
\hline
\forall\, e : Event \bullet \\
\qquad \exists_1\, er : EVENT \bullet e.event = er
\end{array}
$$

The schema condition asserts that each *Event* is uniquely associated with an *EVENT*, or in other words, unique reference semantics. Although this condition ensures the reference semantics, we define a function to map between *EVENT* references and *Event*s to assist in further specification:

$$
\begin{array}{l}
EventRef : EVENT \nrightarrow Event \\
\hline
\forall\, e : Event \bullet \\
\qquad \exists\, er : EVENT \bullet er \mapsto e \in EventRef
\end{array}
$$

Event templates capture the necessary properties of events executed from the template and the information required to establish relationships between events. A set of templates thus defines a program. There can be considerable non-determinism in the relationships specified in a program, typically introduced by logical OR and exclusive OR constructs. Rather than attempt to directly represent the decision trees resulting from such non-determinism, we represent the non-determinism through sets of sets (denoted $\mathbb{P}\,\mathbb{P}\,X$). Each component set represents one possible resolution of the non-determinism. This approach has two key advantages: it simplifies the specification through abstraction, and avoids any suggestion of how such non-determinism should be represented in a programming language.

An event template is thus defined by:

$$
\begin{array}{|l}
\textit{Template} \\
\hline
template : TEMPLATE \\
guards : \mathbb{P}\,\mathbb{P}\,GUARD \\
requires : \mathbb{P}\,\mathbb{P}\,TEMPLATE \\
offers : \mathbb{P}\,\mathbb{P}\,TEMPLATE \\
params : NAME \pfun \mathbb{P}\,PARAMREL \\
\hline
\forall t : Template \bullet \\
\quad \exists_1\, tr : TEMPLATE \bullet t.template = tr \\
\hline
\end{array}
$$

The *guards* are a set of possible combinations of simple guard expressions that, if true, make the whole guard expression true. The *requires* attribute defines the possible combinations of causally preceding events that make this event template executable, and the *offers* attribute defines the possible combinations of causally succeeding events that an event from this template can enable. The significance of these was discussed previously in sections 4.1 and 4.2. The *params* attribute introduces a partial function mapping the set of parameter names to specifications of relationships with other event parameters and environment supplied values. We do not describe these relationships any further because they are captured appropriately in section 4.4. Note that time and location constraints are specified in guards, hence these event attributes are not directly referenced.

Similar to the *Event* specification, we define a function to map template references to schema instances:

$$
\begin{array}{|l}
TempRef : TEMPLATE \pfun Template \\
\hline
\forall t : Template \bullet \\
\quad \exists\, tr : TEMPLATE \bullet tr \mapsto t \in TempRef
\end{array}
$$

4.10.3 Causality Graphs

The behaviour graph representing the history is a relation across a set of events, with each element of the relation an explicit (i.e. required) causal relationship between the paired events. Note that by definition, causality is transitive, but our execution semantics requires a direct representation of the graph and the transitivity can be implied

from the graph. The type of an event graph is thus:

$$
\begin{array}{|l}
\textit{Graph} \\
\hline
events : \mathbb{P}\,Event \\
causes : Event \times Event \\
\hline
\mathrm{dom}\,causes \cup \mathrm{ran}\,causes \subseteq events \\
\hline
\end{array}
$$

A separate *events* set is necessary to allow for events that have no causal relationships (i.e. not connected to any other events by the graph). To assist in further constraining the graph semantics, we define the notion of paths through the graph. A path is defined as a sequence of two or more events connected directly or transitively by the graph relation.

$$
\begin{array}{|l}
paths : Graph \rightarrow \mathrm{seq}\,Event \\
\hline
\forall\, g : Graph;\ path : \mathrm{seq}\,Event \bullet \\
\quad path \in paths(g) \Leftrightarrow \\
\quad\quad \#path = 2 \wedge (head\,path \mapsto last\,path) \in g.causes \vee \\
\quad\quad \#path > 2 \wedge tail\,path \in paths \wedge \\
\quad\quad\quad (head\,path \mapsto head(tail\,path)) \in g
\end{array}
$$

Causality is also irreflexive, so the graph cannot contain cycles. Given our definition of paths, we thus complete the definition by adding an acyclic restriction to the basic *Graph*.

$$CausalGraph == \{g : Graph \mid \nexists p : paths(g) \bullet head\,p = last\,p\}$$

4.10.4 Guards and Time Semantics

Guards on a template must be evaluated in the context of a set of events offering their causality to the template and a time of occurrence. We therefore define a function to represent this guard semantics:

$$satisfied : (GUARD \times \mathbb{P}\,Event \times TIME) \rightarrow \mathbb{B}$$

At this level, we do not need to reason about the actual guard semantics, just note that it is possible to evaluate a guard in the context of these events at the given time.

4.10.5 Correct Execution

We first define the notion of a program:

Program

$templates : \mathbb{P}\,TEMPLATE$
$roots : \mathbb{P}\,\mathbb{P}\,TEMPLATE$

$\bigcup roots \subseteq templates$

The *templates* define the possible events of a program execution and their relationships. The *roots* define a set of possible execution graph roots. A program execution is defined by a causality graph representing its history and a program from which it was executed. We make the set of *templates* in the program an explicit attribute of the schema to simplify subsequent specifications.

Execution

$history : CausalGraph$
$program : Program$
$templates : \mathbb{P}\,TEMPLATE$

$templates = program.templates$
$\forall\, e : history.events \bullet e.template \in templates$

A correct execution is one where all events in the history satisfy the *requires* and *offers* and *guards* attributes associated with their respective templates, and the set of graph roots matches one of the sets specified by program *roots*. The *roots* attribute of a program allows us to prohibit an empty history except where explicitly allowed by a program.

To help us in specification of these constraints, a number of supporting definitions are useful. First an *enables* relation defines if a set of events provides the necessary causality to satisfy the *requires* attribute of a potential successor event, and that the successor event guard is satisfied by this set of predecessors. Note that we include the constraint that the time of the new event must be greater than any of its enabling predecessors.

$$
\begin{array}{|l}
enables : \mathbb{P}\, Event \times Event \\
\hline
\forall\, eset : \mathbb{P}\, Event, e : Event \bullet eset \text{ enables } e \Leftrightarrow \\
\quad \{t : Template \mid \exists\, e' : eset \bullet t = TempRef(e'.template)\} \\
\quad\quad \in TempRef(e.template).requires \land \\
\quad \forall\, e_1 : eset \bullet e_1.time < e.time \\
\quad \forall\, e_1, e_2 : eset \bullet e_1.template \neq e_2.template \land \\
\quad \exists\, gset : e.template.guards \bullet \\
\quad\quad \forall\, g : gset \bullet satisfied(g, eset, e.time)
\end{array}
$$

It is useful to define a stable execution state, that is, one which is incomplete, but does not violate any template constraints. A stable execution state is therefore one that is consistent with the program semantics specified by the template definitions. To support this, we specify an *allows* relation, which defines when an event in the history of an execution allows a subsequent event to use its causality.

$$
\begin{array}{|l}
allows : (Event \times CausalGraph) \times Event \\
\hline
\forall\, e, enew : Event;\; g : CausalGraph \bullet e \mapsto g \text{ allows } enew \Leftrightarrow \\
\quad \exists\, tset : \mathbb{P}\, Template \bullet \\
\quad\quad tset = \{t : Template \mid \exists\, e' : \mathrm{ran}(g.causes \lhd \{e\}) \bullet \\
\quad\quad\quad TempRef(e'.template) = t\} \land \\
\quad\quad TempRef(enew.template) \notin tset \land \\
\quad\quad \exists\, offer : TempRef(e.template).offers \bullet \\
\quad\quad\quad tset \cup \{TempRef(enew.template\} \subseteq offer
\end{array}
$$

This specifies that only a single event from a specific template can use an offer (since iteration allows multiple executions of a template), and that accepting the offer does not invalidate the *offers* constraint associated with template of the preceding event. The existentially quantified *tset* attribute represents the set of offers taken by other successor events in the graph.

A stable execution is thus defined as follows:

$$
\begin{array}{l}
StableExecution == \\
\quad \{ex : Execution \mid \forall\, e : ex.history.events \bullet \\
\quad\quad \mathrm{dom}(ex.history.causes \rhd \{e\}) \text{ enables } e \land \\
\quad\quad \forall\, e' : \mathrm{dom}(ex.history.causes \rhd \{e\}) \bullet e' \mapsto ex.history \text{ allows } e\}
\end{array}
$$

In other words, a stable execution is one in which all events in the execution are correctly enabled, and in which the set of accepted offers associated with each event

does not violate the *offers* attribute of their template.

To define a correct execution, we need to define termination semantics. The *terminates* relation is similar to *allows*, except that it defines if a set of events actually satisfies the *offers* attributes of an event's template:

$$
\begin{array}{|l}
terminates : \mathbb{P}\, Event \times Event \\
\hline
\forall\, eset : \mathbb{P}\, Event;\ e : Event \bullet eset \text{ terminates } e \Leftrightarrow \\
\quad \{t : Template \mid \exists\, e' : eset \bullet t = TempRef(e'.template)\} \\
\qquad \in TempRef(e.template).offers
\end{array}
$$

The set of correct executions is thus defined as follows:

$$
\begin{array}{l}
CorrectExecution == \\
\quad \{ex : StableExecution \mid \\
\qquad \forall\, e : ex.history.events \bullet \operatorname{ran}(\{e\} \lhd ex.history.causes) \text{ terminates } e \wedge \\
\qquad \{t : Template \mid \exists\, e : (ex.history.events - \operatorname{ran} ex.history.causes) \bullet \\
\qquad\quad TempRef(e.template) = t\} \in ex.program.roots\}
\end{array}
$$

In other words, the set of correct executions are the stable executions where all events in the history are also correctly terminated, and the where roots of the history graph satisfy the program *roots* specification. Note that this does not guarantee that a program will terminate, just that if it does terminate in a state that satisfies the axiom, then the execution is correct. A non-terminating program can thus never have a correct execution, but can have a stable execution.

4.10.6 Transition Semantics

We define a transition relation over executions that specifies the valid transitions that can occur in program execution. Informally, a transition represents the addition of a set of events and arcs to a graph to yield a new graph. Executing transitions that satisfy this relation ensures that each step of the execution is a *StableExecution*, or in other words, it satisfies the program semantics encoded in the template definitions.

We begin by specifying a graph transition relation:

$$
\begin{array}{|l}
GraphTransition : CausalGraph \times CausalGraph \\
\hline
\forall g, g' : Graph \bullet g \mapsto g' \in GraphTransition \Leftrightarrow \\
\quad g.events \subset g'.events \land \\
\quad \exists\, eset : \mathbb{P}\, Events \mid eset = g'.events - g.events \\
\quad\quad \forall (e_1 \mapsto e_2) : g'.causes - g.causes \bullet e_1 \notin eset \land e_2 \in eset \land \\
\quad\quad \forall e_1, e_2 : eset \bullet e_1.template \neq e_2.template \\
\quad\quad \forall e : eset \bullet \exists\, preds : \mathbb{P}\, g.events \mid preds = g'.causes \rhd e \\
\quad\quad\quad preds \text{ enables } e \land \\
\quad\quad\quad \forall e' : preds \bullet e' \mapsto g' \text{ allows } e
\end{array}
$$

This states that the transition can only add relations (arcs) to the graph and all arcs added must be from old events to new events. This guarantees that each transition makes progress, and that extant causal relationships between events (or lack thereof) are immutable. The subsequent predicates specify that only one event from a given template may be executed in a single transition, that each new event must be enabled by the events that immediately precede them in the new graph, and that the new graph must allow each new event to accept causality from those predecessors.

The existentially quantified *eset* attribute defines the set of events added to the graph. This use of a set of events models true concurrency, where multiple events can execute independently and concurrently. We do not constrain the execution time of these events, meaning that the events can be logically concurrent and hence executed at any time, noting that we do not allow causal relationships within the set.

The restriction preventing multiple events from the same template executing concurrently allows us to specify successors using template names, and also simplifies the definition of programming language semantics, as will be seen in subsequent chapters of the thesis.

A transition from one *Execution* to another in a program is thus defined as follows:

$$
\begin{array}{|l}
Transition : Execution \times Execution \\
\hline
\forall ex_1, ex_2 : Execution \bullet (ex_1 \mapsto ex_2) \in Transition \Leftrightarrow \\
\quad ex_1.program = ex_2.program \\
\quad (ex_1.history \mapsto ex_2.history) \in GraphTransition
\end{array}
$$

We propose that any valid *Transition* from a *StableExecution* leads to another *StableExecution.*

Proof: given the definition of *StableExecution* in section 4.10.5 the proposition can only fail if:

1. there exists an event e in the range of the *causes* relation that is not correctly enabled; or

2. there exists an event e in the range of the *causes* relation for which the set of enabling events does not *allow* e.

If we begin with a stable execution, the first condition is impossible due to the $(preds \,\text{enables}\, e)$ constraint on all events added by a *GraphTransition.* Similarly, the second condition is made impossible by the $(e' \mapsto g' \,\text{allows}\, e)$ constraint.

It is not possible to guarantee termination of all programs, thus a *CorrectExecution* cannot be guaranteed. The execution of a sequence of valid *Transition* as defined by the axiom above, however, ensures that the program is being executed in a manner consistent with the program specification. If finite termination is always possible for a given program, then continued execution of valid *Transitions* will eventually lead to a *CorrectExecution.*

4.11 Formalising Distribution

The preceding sections have formalized the execution of programs based on the *Finesse* behavioural model. Our informal description of the distribution semantics describes the conditions under which the execution of a program can be distributed across a set of participants. This section formalizes the distribution of this execution, showing that subject to some program constraints, distributed participants can correctly execute locally enabled *Transitions* against a partial view of the execution history.

The fundamental aspect of the distributed semantics is the notion of location. Each state machine participating in the distributed execution of a program has a unique location and maintains a view of the execution history. The execution history of each machine is populated with all locally executed events and any remote events about which it has been notified through direct or indirect communication. As previously specified, a valid transition involves the addition (execution) of a set of events to the

history graph. A valid transition must also ensure that the relevant *allows* and *offers* relationships are satisfied, and that only a single event from any template is involved in a transition. If we show that the execution of a causally unrelated (logically concurrent) remote event cannot invalidate the execution of a locally enabled event, then the local event can be executed regardless of any concurrent remote behaviour. Note here that "concurrent" means all behaviour that occurs at each remote location before the next event notification from that remote location. The following subsections formally capture this approach.

4.11.1 Templates and Locations

We first define a distributed program as one that associates the templates of a program with specific locations:

DistProgram

$program : Program$
$roles : Template \pfun LOCATION$

$program.templates = \operatorname{dom} roles$

A distributed execution that has a set of executions of the program (*participants*), each bound to a location, and constrains all program templates to that same set of locations:

DistExecution

$participants : \mathbb{P}\, Execution$
$dprogram : DistProgram$
$interfaces : Execution \pfun LOCATION$

$participants = \operatorname{dom} interfaces \land$
$ran(interfaces) = ran(dprogram.roles) \land$
$\forall p : participants \bullet p.program = dprogram.program$

The binding of templates to locations allows the participating executions to assert that an event associated with a template cannot be executed concurrently in another locations, thus preventing any attempts to execute multiple events from the same template in a single transition. The binding of template to location can be performed at

program initialization, although for tractability it would be usual for the co-location of events to be statically defined. This co-location of events is typically associated with the notion of an *interface*. Note that it is also possible for a set of participants to exhibit similar behaviour by copying and renaming a set of templates and distinguishing their location at initialization time or even at run time based on a statically defined set of locations for a nominated behaviour. This corresponds nicely to the notion of a shared *role* or *process group* in a distributed computation.

4.11.2 Initialization

The key requirement for distributed execution is that all participants start from the same state:

$$InitialState == \{d : DistExecution \mid \forall p : d.participants \bullet p.history = \varnothing\}$$

4.11.3 Distributed State

We now need a definition for the state of a distributed execution. This is defined as the execution of the shared program with the history defined as the union of the local execution histories. We first define the merge of a set of *CausalGraph* structures:

$$
\begin{array}{|l}
GraphMerge : \mathbb{P}\, CausalGraph \nrightarrow CausalGraph \\
\hline
\forall gset : \mathbb{P}\, CausalGraph, g : CausalGraph \bullet GraphMerge(gset) = g \Leftrightarrow \\
\quad g.events = \bigcup\{eset : \mathbb{P}\, Event \mid \exists g : gset \bullet eset = g.events\} \land \\
\quad g.causes = \bigcup\{erel : \mathbb{P}(Event \times Event) \mid \exists g : gset \bullet erel = g.causes\}
\end{array}
$$

Using this definition, we define the distributed state:

$$
\begin{array}{|l}
DistState : DistExecution \nrightarrow Execution \\
\hline
\forall d : DistExecution, ex : Execution \bullet DistState(d) = ex \Leftrightarrow \\
\quad ex.history = GraphMerge \\
\qquad (\{g : CausalGraph \mid \exists ex' : d.participants \bullet ex'.history = g\})
\end{array}
$$

The distributed state axiom captures the notion that the state of a distributed execution is the execution state formed by the merge of the local states in the execution.

4.11.4 Distributed Transitions

We define a distributed transition function as a set of local transitions with the execution of any template restricted to the participant with the same location as that template. Formally:

$$
\begin{array}{l}
DistTransition : DistExecution \times DistExecution \\
\hline
\forall\, d, d' : DistExecution \bullet d \mapsto d' \in DistTransition \Leftrightarrow \\
\quad d.dprogram = d'.dprogram \wedge \\
\quad \exists\, ex \mapsto l : d.interfaces, ex' \mapsto l' : d'.interfaces \bullet \\
\quad\quad l = l' \wedge ex \mapsto ex' \in Transition \wedge \\
\quad \forall\, p : d.participants \bullet \exists\, p' : d'.participants \bullet \\
\quad\quad p = p' \vee \\
\quad\quad (d.locations(p) = d'.locations(p') \wedge \\
\quad\quad\quad p \mapsto p' \in Transition \wedge \\
\quad\quad\quad \forall\, e : (p'.history.events - p.history.event) \bullet \\
\quad\quad\quad\quad d'.dprogram.roles(e.template) = d'.interfaces(p'))
\end{array}
$$

Note that the specification includes the constraint that at least one local transition must have occurred, thus ensuring progress is made.

To show the validity of a distributed execution, we must show that if we start from a *DistState* that satisfies the *StableExecution* axiom, a valid *DistTransition* results in a *DistState* that also satisfies *StableExecution.* This can be proven if we show that any local *Transition* of a participant *Execution* is also a valid *Transition* of the distributed execution state. Reiterating the argument given in section 4.10.6, the proposition can only fail if, after a local transition:

1. there exists an event e in the range of the *causes* relation of the *DistState* that is not correctly enabled; or

2. there exists an event e in the range of the *causes* relation of the *DistState* for which the set of enabling events does not *allow* e.

The first condition is a constraint over the set of enabling arcs for the event e in the distributed state. The location constraint on template execution in *DistTransition* ensures that only one participant will add e to its history, hence only that participant will add enabling arcs for e to the graph. Thus, if the local execution correctly satisfies

the $(eset \text{ enables } e)$ constraint of a *Transition*, the constraint will be satisfied in the distributed state since the *eset* associated with e cannot be modified by the merge of participant histories.

The second condition cannot be satisfied in the same manner: it is quite possible that distinct events at different participants will use an offer from a single enabling event and this could violate the *allows* condition on the enabling event in the distributed state, but does not necessarily violate the *allows* condition on the local state.

A distributed program has an *allows* conflict if any two sets of possible offers a template can make have mutually exclusive members that are not co-located. As discussed in the informal description of this problem in section 4.6, we address this issue by defining the set of safe programs as those that do not have this conflict, or have guards to ensure that such conflicts can never occur in a distributed execution. More formally:

$$
\begin{array}{l}
\mathit{SafeProgram} : \mathbb{P}\, \mathit{DistProgram} \\
\hline
\forall p : \mathit{DistProgram} \bullet p \in \mathit{SafeProgram} \Leftrightarrow \\
\quad \forall d : \mathit{DistExecution} \mid d.\mathit{dprogram} = p \bullet \\
\quad\quad \mathit{DistState}(d) \in \mathit{StableExecution} \land \\
\quad\quad \exists e : \mathit{DistState}(d).\mathit{history}.\mathit{events} \bullet \\
\quad\quad\quad \exists \mathit{tset}_1, \mathit{tset}_2 : e.\mathit{offers}, t_1, t_2 : \mathit{Template} \bullet \\
\quad\quad\quad\quad p.\mathit{roles}(t_1) \neq p.\mathit{roles}(t_2) \\
\quad\quad\quad\quad t_1 \in \mathit{tset}_1 \land t_2 \notin \mathit{tset}_1 \land \\
\quad\quad\quad\quad t_2 \in \mathit{tset}_2 \land t_1 \notin \mathit{tset}_2 \\
\quad\quad\quad \Rightarrow \nexists \mathit{eset}_1, \mathit{eset}_2 : \mathbb{P}\, \mathit{DistState}(d).\mathit{history}.\mathit{events}, e_1, e_2 : \mathit{Event} \bullet \\
\quad\quad\quad\quad e_1.\mathit{template} = t_1 \land e_2.\mathit{template} = t_2 \land \\
\quad\quad\quad\quad e \in \mathit{eset}_1 \land e \in \mathit{eset}_2 \land \\
\quad\quad\quad\quad \mathit{eset}_1 \text{ enables } e_1 \land \mathit{eset}_2 \text{ enables } e_2
\end{array}
$$

This specification is somewhat complex, but essentially says that if a conflict exists in an execution of a program because of mutually exclusive offers from an event e, then there cannot be sets of enabling events in the execution state including e that enable those mutually exclusive events. Based on the definition of the *enables* relation in section 4.10.5 this implies that either the guards on those events cannot be satisfied by that set of enabling events or the offer cannot be in the *requires* set of one of the events. In general, we expect that the safety of programs will be guaranteed by ensuring mutually exclusive events have similarly exclusive guard conditions.

Returning to the discussion of the satisfaction of our *allows* constraint, we can guarantee this constraint if the distributed program is a *SafeProgram*, and thus, any valid local transition in a distributed execution will result in a *StableExecution* state. From this we assert that any *DistTransition* against a *DistExecution* in a *StableExcution* state will result in another *StableExecution* state.

4.11.5 History Updates

One of the issues not addressed in the above discussion is the updating of local execution histories to reflect remote events in the distributed execution state. Without this, remote events will never be visible in local histories and hence local events with dependencies on those remote events cannot be executed. We define history updates formally as follows:

$$
\begin{array}{l}
\mathit{HistoryUpdate} : \mathit{DistExecution} \times \mathit{DistExecution} \\
\hline
\forall d_1, d_2 : \mathit{DistExecution} \bullet (d_1, d_2) \in \mathit{HistoryUpdate} \Leftrightarrow \\
\quad \mathit{DistState}(d_1) = \mathit{DistState}(d_2) \wedge \\
\quad d_1.\mathit{dprogram} = d_2.\mathit{dprogram} \wedge \\
\quad \forall p_1 : d_1.\mathit{participants}, p_2 : d_2.\mathit{participants} \bullet \\
\quad\quad d_1.\mathit{locations}(p_1) = d_2.\mathit{locations}(p_2) \Rightarrow \\
\quad\quad\quad p_1.\mathit{history.events} \subseteq p_2.\mathit{history.events} \wedge \\
\quad\quad\quad p_1.\mathit{history.causes} \subseteq p_2.\mathit{history.causes} \wedge \\
\quad \exists p_1 : d_1.\mathit{participants}, p_2 : d_2.\mathit{participants} \bullet \\
\quad\quad d_1.\mathit{locations}(p_1) = d_2.\mathit{locations}(p_2) \wedge \\
\quad\quad p_1.\mathit{history.events} \subset p_2.\mathit{history.events}
\end{array}
$$

A key aspect of this specification are that the distributed state is unchanged: only the local execution histories are updated with events from remote participants. This means that the *HistoryUpdate* transition cannot influence the *StableExecution* property of a distributed execution. We also require that at least one history is updated in the transition.

4.11.6 Properties of Transitions

The *DistTransition* and *HistoryUpdate* transitions define the possible state changes in the execution of a distributed program and guarantee that, given an initially empty execution history and a safe program, all reachable states are *StableExecution* states.

There are some key properties of this specification that are worth noting:

1. The transition semantics does not require that each local execution is a *StableExecution*, thus local histories can be sparse views of the distributed state provided all transitions satisfy the *DistTransition* and *HistoryUpdate* specifications.

2. The *DistTransition* and *HistoryUpdate* are, by definition, mutually exclusive because the former explicitly modifies the *DistState* and the other explicitly does not modify *DistState*. Thus, for any distributed execution there must be a logically serialized set of *DistTransition* and *HistoryUpdate* transitions.

3. Despite the need for logical serialization, the *DistTransition* and *HistoryUpdate* are both specified as independent transitions across a set of local executions. This very nicely captures the logical concurrency of the system: providing there are no intervening *HistoryUpdate* transitions, any set of transitions from each participant individually satisfying the *DistTransition* specification can be combined into a single, logically concurrent, transition. Similarly for local *HistoryUpdate* transitions.

We have now formally defined the Finesse execution semantics, first for a centralized execution and then for a distributed execution. This specification captures the necessary properties of such executions and their programs.

4.12 Concluding Remarks

The informal and formal descriptions presented in this chapter define a distributed, asynchronous execution model for parallel programs. As discussed in the introduction to this chapter, there is no other equivalent, executable model in widely published literature, so this result is quite significant in itself. Petri nets can provide a similar semantics but impose additional synchronization requirements and provide a less approachable programming model.

Subsequent chapters use this model in the context of the binding, role and interface architectural concepts introduced in chapter 2, but this execution model stands alone and can be used in many parallel or distributed programming contexts.

4.13 Acknowledgments

Some of the work set out in the chapter is derived in from ideas proposed by this author but developed and refined in collaboration with Andry Rakotonirainy, Stephen Crawley and Zoran Milosevic. I would like to express my sincere thanks to them for their efforts. This early work was published in a paper presented at HICSS'97[103] and subsequently republished in the British Computer Journal[102].

I would also like to thank Colin Fidge for his insights and feedback in developing this aspect of the work, and a thank you to Anthony MacDonald for his assistance in reviewing the Z specification.

Chapter 5

Language

5.1 Introduction

The preceding chapter provided a semantic model for the execution of asynchronous distributed programs. This chapter[1] describes the *Finesse* language, which can use the semantic model as the basis for execution. The *Finesse* language, while not the only possible application of this model, provided the impetus for development of the semantic model and is an example of how the semantic model can be presented in a useful language.

The *Finesse* language is perhaps best described as a coordination language and a description of the language has previously been published in that research sphere[9]. The syntax and structuring concepts of the language, however, have grown from the needs of open distributed processing and in particular, the A1$\surd$ model[11]. The language is intended to describe the behaviour and coordination of autonomous software components in an open, distributed environment. As such, it bears only a minimal resemblance to existing programming language syntax, including interface definition languages like CORBA IDL.

While *Finesse* provides similar structuring and high-level functionality to other coordination languages, it also differs in many ways. In particular:

1. *Finesse* abstracts over communication, allowing transformation of data and com-

[1]This chapter is derived from a paper presented at the Symposium of Applied Computing in February 1998[9]

piler or run-time optimization of message passing between components.

2. *Finesse* includes a representation of time, allowing the specification of quality of service properties;

3. *Finesse* is independent of the language used for programming the distributed components. It is similar in concept to CORBA IDL, for example, where the program is compiled to produce interface *stubs* for components in the chosen language(s);

In this chapter, we present the syntax of *Finesse* and describe the relationship of the syntactic constructs to the semantic model described in the preceding chapter. We build on the language overview of chapter 2 and the semantic model, so a basic understanding of the concepts presented in those chapters is assumed. In particular, we assume that the key concepts of *binding*, *role*, *interface* and *event relationships* are understood. A BNF specification of the language syntax is given in appendix A.

5.2 Basic Syntax and Structure

5.2.1 Structure of a *Finesse* Program

A *Finesse* program, also called a *binding* has an outer scope introduced by the keyword **Binding** and the name of the binding. This outer scope defines the program boundary: all behaviour present in the binding must be described in this scope. A set of **Import** statements may appear at the beginning of this scope. An **Import** statement identifies another binding program whose role and interaction definitions may be referenced and re-used in this binding.

This opening is followed by two sections defining roles and interactions. The **Roles** section defines the required behaviour of participating components, and the **Interactions** section defines the relationship between events at different roles. Braces (`{...}`) are used to delimit the scope of definitions. Note that in the following examples, ellipses (`...`) are used to avoid including unnecessary detail and are not a syntactic construct. The basic structure is thus:

```
Binding Example {
  Import ...;
  Roles {
    ...
  }
  Interactions {
    ...
  }
}
```

5.2.2 Describing Roles

A binding has one or more role definitions, introduced by a role name. A role definition can be prefixed with a cardinality constraint enclosed in square braces `[ ]`, which constrains the number of components that can fill a role in single a binding instance (program execution). The place-holder # represents the actual cardinality. Where no cardinality constraint is given, the default cardinality is exactly one, for example:

```
Roles {
  Client { ... }
  [#>=1] Server { ... }
}
```

This specifies that there are two roles, *Client* and *Server* and that there is exactly one *Client* and at least one *Server* in the binding.

Behaviour described within a role takes the form of event relationship specifications. In terms of the semantic model, these specifications take the form of event template definitions connected by a number of relationship operators and modifiers. These operators and modifiers are discussed further in subsequent sections. Events executed from the event templates within a role specification are required to be co-located. Where multiple interfaces can fulfill the role, each interface executes the described behaviour independently (i.e. a logical AND of the behaviours) except where linked by specifications in the **Interactions** section.

5.2.3 Describing Interactions

The **Interactions** specification defines relationships between event templates occurring in the roles. In this section, event templates are referred to by the role name,

followed by a period '.' and the event template name. This reference to an event template can also have a cardinality constraint to deal with situations where multiple components fill the role. For example:

```
Binding Example {
  Import ...;
  Roles {
    Client { send! }
    [#>=1] Server { receive? }
  }
  Interactions {
    Client.send -> [#=all] Server.receive
  }
}
```

The place-holder `#` in the **Interactions** specification refers to the number of components executing the event template, while the place-holder `all` refers to the number of components filling the role. In the above example, the client role executes a `send` event followed by all servers executing the `receive` event. In other words, this binding is a high-level description of reliable multicast. Behaviour described in the interactions section cannot introduce new event templates: it can only use event template names defined in the roles section.

5.2.4 Event Templates

The behaviour within roles and interactions is defined by event templates and their relationships. An event template is introduced in the roles section by a name, a direction indicator, and a parameter list, for example:

```
e!(x:t1, y:t2)
```

where `e` is the event template name, `!` indicates that it is an output event, `x,y` are the event parameters, and `t1,t2` are the data types of the parameters. Events are uni-directional, that is, they can be input events or output events but not both. The `?` character is used in place of the `!` to indicate an input event. Input and output are relative to the role, that is, an output event implies that the component filling the role is providing parameter values, while an input event implies that program execution is

providing parameter values. In terms of the semantic model, the direction indicator is unnecessary because all events imply synchronization between the interface implementing a role and the program execution. Direction indicators are included in the language, however, because there is typically a clear distinction between input events and output events in component implementations, for example, the input and output associated with a method call.

The direction indicator and parameter list are only included in the first definition of an event template. This means they may only appear in role definitions. Where a template name appears more than once in a role definition, only the first can include these annotations.

5.2.5 Named Behaviours

Roles can contain named behaviours that group together a set of event templates and allow the **Interactions** section to refer to some subset of the role when defining interaction behaviour, for example:

```
Binding {
  Roles {
    Client {
       read { send! -> receive? } ->
       write { send! -> receive? }
    }
    ...
  }
  Interactions {
    Client.read ...
  }
}
```

Named behaviours define a scope for event template names, allowing the role definitions to re-use template names in a different named behaviour when appropriate. Reference to such event templates in the interactions section must include the role and any scope names, e.g. using the example above `Client.read.send`.

5.3 Describing Behaviour

The constructs of the preceding subsections have hinted at how behaviour is described in *Finesse* but are primarily aimed at describing program structure and event attributes. In this section, we describe how the operators and modifiers of *Finesse* allow us to describe event relationships in terms of the semantic model presented in 4.

5.3.1 Introducing the Causality Operator

Causality between event templates is introduced using the –> operator. The statement `e1 –> e2` has implications for both the left hand and right hand templates. In terms of the execution semantics:

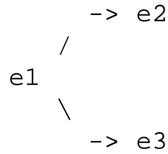
$$\begin{array}{l} e1 \text{ –> } e2 \Rightarrow \\ \quad e1 \in e2.\mathit{requires} \wedge \\ \quad e2 \in e1.\mathit{offers} \end{array}$$

In other words, the statement implies that the template `e2` requires an offer from an `e1` event before it can be executed, and that an `e1` event offers its causality to an `e2` template. This defines the causality operator in its simplest form where there is no non-determinism and the templates `e1` and `e2` refer to single occurrences of events (cardinality of exactly one).

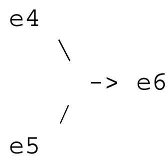
Instances of such causal relationship specifications can be chained, that is, `e1 –> e2 –> e3` without any change to the semantics. Considerably more complex expressions can be used on either side of the operator, however. In subsequent discussion, we will refer to the *LHS* and *RHS* meaning the expression to the left and to the right of the operator respectively.

5.3.2 Complex Expressions

Expressions on either side of the causality operator can refer to a graph of events, for example, as a result of using named behaviours. In other words, the LHS of the –> operator could be the graph:



In this case, the operator applies to each leaf template of the graph, that is, `e2` and `e3` in this case. Similarly, if the RHS of the operator is a graph with multiple root templates, the operator applies to each roots. Where there are multiple roots and leaves, the operator applies to the product of leaves and roots (i.e. each possible pairing), for example, if we use the above graph as our LHS and our RHS were:



We have:

$$
\begin{aligned}
&\mathit{LHS} \mathrel{-\!\!>} \mathit{RHS} \Rightarrow \\
&\quad e2 \mathrel{-\!\!>} e4 \wedge \\
&\quad e2 \mathrel{-\!\!>} e5 \wedge \\
&\quad e3 \mathrel{-\!\!>} e4 \wedge \\
&\quad e3 \mathrel{-\!\!>} e5 \wedge
\end{aligned}
$$

5.3.3 Logical AND

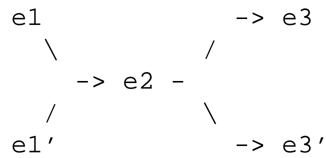
The logical AND of two behaviours, `LHS AND RHS`, implies that both behaviours must occur for correct execution of the program. Where there are no common, fully-qualified template names in the LHS and RHS behaviours, the two behaviours are independent: there are no causal relationships and they can execute concurrently. Where any template names match, the LHS and RHS must synchronize on the matching event templates, or in other words, common event templates join the two behaviours. For example:

```
e1 -> e2 -> e3

    AND

e1' -> e2 -> e3'
```

implies:

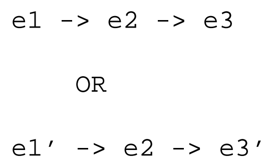


In terms of the offers and requires, it implies:

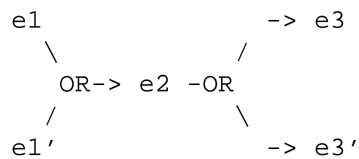
$$\{e1, e1'\} \subseteq e2.\mathit{requires} \wedge$$
$$e2 \in e1.\mathit{offers} \wedge$$
$$e2 \in e1'.\mathit{offers} \wedge$$
$$e2 \in e3.\mathit{requires} \wedge$$
$$e2 \in e3'.\mathit{requires} \wedge$$
$$\{e3, e3'\} \subseteq e2.\mathit{offers}$$

5.3.4 Logical OR

The logical OR of two behaviours `LHS OR RHS`, implies that either or both behaviours can occur. As with AND, the behaviours must synchronize on common event template names, that is:



implies:



In terms of offers and requires:

$$(e1\ \mathit{OR}\ e1') \in e2.\mathit{requires} \wedge$$
$$e2 \in e1.\mathit{offers} \wedge$$
$$e2 \in e1'.\mathit{offers} \wedge$$
$$e2 \in e3.\mathit{requires} \wedge$$
$$e2 \in e3'.\mathit{requires} \wedge$$
$$(e3\ \mathit{OR}\ e3') \in e2.\mathit{offers}$$

It becomes clear from this example that the *requires* and *offers* associated with an event template are decision trees, as described in the execution semantics. Any level of nesting of such logical expressions is possible.

5.3.5 Exclusive OR

The exclusive OR of two behaviours `LHS XOR RHS`, implies that one or other of the two behaviours must occur but not both. Since the behaviours are mutually exclusive, no synchronization is possible or necessary. Using our same example:

```
e1 -> e2 -> e3

    XOR

e1' -> e2 -> e3'
```

implies:

$$
\begin{aligned}
&(e1\ XOR\ e2) \in e2.requires \wedge \\
&e2 \in e1.offers \wedge \\
&e2 \in e1'.offers \wedge \\
&e2 \in e3.requires \wedge \\
&e2 \in e3'.requires \wedge \\
&(e3\ XOR\ e3') \in e2.offers
\end{aligned}
$$

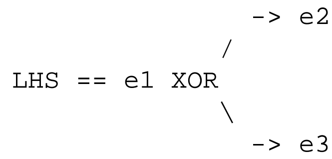
The XOR operator can also be used to capture optional behaviour, that is:

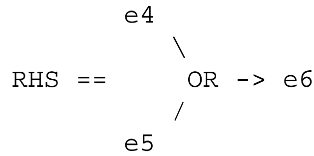
$$
\begin{aligned}
&e1 \rightarrow e2\ XOR\ e1 \Rightarrow \\
&\quad e1 \in e2.requires \wedge \\
&\quad (e2\ XOR\ \varnothing) \in e1.offers \\
&e1 \rightarrow e2\ XOR\ e2 \Rightarrow \\
&\quad (e1\ XOR\ \varnothing) \in e2.requires \wedge \\
&\quad e2 \in e1.offers
\end{aligned}
$$

5.3.6 Combining Logical Operators and Complex Expressions

In the preceding subsections we have described logical operators and complex expressions in isolation. From the discussion, it is clear that the non-determinism in predecessor and successor event templates caused by logical OR and XOR operators must be captured in the *requires* and *offers* of an event template. In the presence of complex

expressions, it follows that any non-determinism involving leaf nodes of a LHS must be captured in the *requires* of the RHS root nodes and any non-determinism involving the RHS root nodes must be captured in the *offers* of the LHS leaf nodes. For example, if we have:





We get:

$$
\begin{aligned}
&\mathit{LHS} \;\text{–>}\; \mathit{RHS} \Rightarrow\\
&\qquad (e2\ \mathit{XOR}\ e3) \in e4.\mathit{requires} \wedge\\
&\qquad (e2\ \mathit{XOR}\ e3) \in e5.\mathit{requires} \wedge\\
&\qquad (e4\ \mathit{OR}\ e5) \in e2.\mathit{offers}\\
&\qquad (e4\ \mathit{OR}\ e5) \in e3.\mathit{offers}
\end{aligned}
$$

The notion of the leaf or root node is essentially replaced with the notion of a leaf or root *expression* whenever there is non-determinism introduced by OR or XOR across nodes.

5.3.7 The Logic of Cardinality Constraints

Cardinality constraints specified in interaction specifications imply a “subset choice” semantics: a correct behaviour executes a subset of the possible events where the subset satisfies the specified cardinality constraint. Such specifications take the form:

```
R1.e1 -> [# > n] R2.e2
```

where $\#$ refers to the number of component interfaces in role $R2$ that actually execute $e2$, $>$ is a numeric comparison operator (i.e. it could be a $<$, $=$ etc), and n is a numeric

value. The cardinality constraint imposes a logical relation over the distinct templates that could be executed at each of the interfaces filling the roles.

The effect of the cardinality constraint is equivalent to a logical XOR of all possible subsets that satisfy the cardinality constraint. For example, if we have three interfaces $I1$, $I2$ and $I3$ in role $R2$ and the cardinality constraint is $[\# = 2]$, we have:

$$(I1.e2 \text{ AND } I2.e2) \text{ XOR} (I2.e2 \text{ AND } I3.e2) \text{ XOR} (I1.e2 \text{ AND } I3.e2)$$

This can be considered a rewriting rule over the original expression, thus the expression becomes:

```
R1.e1 -> I1.e2          R1.e1 -> I2.e2          R1.e1 -> I1.e2

     AND         XOR         AND         XOR         AND

R1.e1 -> I2.e2          R1.e1 -> I3.e2          R1.e1 -> I3.e2
```

The semantics of these logical operators specified in the preceding sections can then be applied. The usual mathematical reduction and/or simplification of logical expressions involving AND, OR and XOR can be applied. Note here that event templates specified in role and interaction definitions must be annotated with an interface identifier at binding instantiation to distinguish them in the program execution graph, as discussed in section 4.7 of the previous chapter.

The implications of cardinality constraints for non-determinism are significant and it is worth noting that this example and others can potentially cause problems for implementation. By definition, the behaviour associated with a set of interfaces implementing roles is distributed. Cardinality constraints applied to a role can therefore result in distributed, mutually exclusive successors to an event. Such mutual exclusion occurs whenever a maximum cardinality less than the number of interfaces filling a role is specified on the RHS of the causality operator. Consider the example above: a distributed decision must be made to exclude either $I1.e2$, $I2.e2$ or $I3.e2$ to ensure that the constraint is satisfied. This problem is discussed at some length in section 4.6 of the preceding chapter. The recommended solution is to require that event template guard on any RHS expression guarantees the necessary mutual exclusion.

The *Finesse* language specification does not implement this recommendation through

its syntax, however, compilers should flag any program construct that could potentially lead to such problems. In the case of cardinality constraints, there are also builtin boolean functions (discussed in subsequent sections) that can be used to guard such behaviour.

5.4 Control Flow

The preceding section provided a basis for describing event relationships using a number of operators. In this section, we introduce control flow constructs, and in particular, constructs for guards and iteration.

5.4.1 Roles, Interactions and Iteration

We know intrinsically that iterative behaviour must be implemented by the components participating in a distributed program. Connecting the iterative behaviour of distinct components in a well-defined and clear specification is difficult at best. In *Finesse*, we try to minimize this complexity by allowing iteration constructs only in role specifications: the interactions section of a specification can thus only be used to "connect" the iterative behaviours and not introduce any new iteration. This is in keeping with the goal of being a language aimed at connecting components. As can be seen from the previous and subsequent example programs, this keeps the interaction specifications relatively simple, and loses little or no expressive power.

Revisiting the semantics of the −> operator, we add the assertion that in the interactions specification, the expression `LHS −> RHS` implies that *any* instance of the `LHS` behaviour is followed by a distinct instance of the `RHS` behaviour, subject to any non-determinism introduced by logical operators. This means that the relationship applies distinctly to any iterated instances of the `LHS` and `RHS` behaviours.

In the *Finesse* language, we do not allow parallel iteration within a role. This avoids the inherent problem of connecting related parallel behaviours across roles, but prevents a role from creating independent threads. Parallel instances of a behaviour can be introduced by allowing multiple instances of a role: this explicitly identifies each instance and the connections between these behaviours and others. It can also be

introduced by using explicitly and statically named copies of an imported behaviour. While it would be possible to extend the semantic model and language to include parallel iteration within a role, many programs can be expressed without such iteration as is demonstrated in chapter 8. Future versions of the language might allow the dynamic creation of subordinate bindings to address this problem and avoid the issues related to connecting parallel behaviours across roles.

5.4.2 Iteration

Iteration is introduced by a `while` construct:

```
while [guard] { LHS }
-> RHS
```

This is equivalent to:

```
[guard] LHS -> { [guard] LHS XOR [NOT guard] RHS }
XOR
[NOT guard] RHS
```

We require, however, that expressions of the form `LHS -> LHS` are not permitted in *Finesse*. This is a syntactic restriction aimed at clearly distinguishing iteration from sequential behaviour and avoiding ambiguity when an expression appears more than once in a specification. It makes little difference to the language semantics, as suggested by our definition of the while construct semantics above.

We also allow an unguarded iteration construct:

```
loop { LHS }  -> RHS
```

This is equivalent to:

```
LHS -> { LHS XOR RHS }
XOR
RHS
```

In other words, this construct allows the LHS behaviour to be executed zero or more times, but the RHS behaviour can be chosen at the end of any iteration. The choice of the RHS behaviour terminates the loop, and the RHS behaviour must be immediately distinguishable from the LHS behaviour.

5.5 Resolving Event References

Throughout a *Finesse* program it is necessary to make references to other events that have occurred. The nature of role/interaction behaviour specifications is such that while these references identify a single event template definition from a role, they can identify multiple event instances. This is due both to iteration (multiple instances of the event executed by a single participant) and roles with greater than one participant (multiple instances of the event executed by different participants). The syntax also allows a **prev** keyword for referencing the immediately preceding event. In order to define the language syntax, we must specify how these event references resolve to particular events.

Within a role specification, all references resolve to the nearest causally preceding, co-located event that matches the reference. The term "nearest" means that there is no other causally succeeding event with the same name. References in role specifications should not include a role name. Since role behaviour cannot include parallel iteration, this rule ensures that the reference can only resolve to a single event. The **prev** reference always identifies the event specified immediately to the left of the current `->` operator. The **prev** keyword is invalid where the LHS is an expression with multiple tail events.

Within an interaction specification, the reference resolves to the nearest causally preceding event that matches the reference. Where multiple concurrent events are identified by this rule (i.e. there is no single matching event that is causally after all other matching events), a co-located event is given preference, then an event is randomly chosen. The **prev** references identifies the event specified immediately to the left of the current `->` operator. If this reference is of the form `Role.event` with a cardinality greater than one, then a single event can be randomly chosen from the set.

The **prev** keyword is invalid where the LHS is an expression with multiple distinct tail events.

Note that this interpretation of event references is language specific. The semantic model described in the preceding chapter is quite able to accommodate an alternative syntax that uses event references to identify sets of events, for example.

5.6 Guards

A guard is a logical expression that must be satisfied before an event can be executed. This is in addition to any cardinality or causal predecessor constraints. In the *Finesse* language, guards correspond exactly to the guard semantics presented in the previous chapter. They are introduced using the following syntax:

```
[guard] LHS
```

where `guard` is a logical expression and `LHS` is any behaviour expression. As with the `RHS` of our behavioural operators, the guard applies across all root nodes of the expression. Where an event template has both a guard and a cardinality constraint, they must be contained within the same square braces and joined by a logical AND operator. While this is slightly inconsistent semantically, it makes considerable sense syntactically since the cardinality is a restriction that must be satisfied when executing a set of events.

A named expression might appear in several parts of a specification, requiring synchronization of all instances. Distinct guards can be applied to any instance of the expression, with the result being the logical AND of all guard expressions.

We defer the detailed syntax of guard expressions to the implementation, since it will depend on the set of functions and operators applicable to the supported data types. It is required, however, that all implementations support a functional guard of the form $f(x1, x2, ...)$ and the logical operators `AND`, `OR`, `XOR` and `NOT`. A number of explicit guard functions associated with time and ordering constraints are specified in the following subsections. Note that the syntax of cardinality constraints is specified previously in section 5.3.7.

5.6.1 Time Guards

Timing constraints are included in *Finesse* programs through guards and the provision of two built-in functions: **timeless** and **timemore**. These functions take an event reference and a real-valued expression (which could itself be a function). **timeless** evaluates to true if the number of seconds since the referenced event is less than the supplied value. Conversely, **timemore** evaluates to true if the number of seconds since the referenced event is equal to or greater than the supplied value. For example:

```
e1!() -> [timeless(e1, 10.0)] e2?()
```

This specifies that the event `e2` must occur within 10.0 seconds of `e1`. As specified in the semantic model, guards involving time are only permitted to compare time deltas. Literal time deltas are represented as a real number indicating a number of seconds. Implementations of *Finesse* should allow for clock skew when evaluating time guards involving events at different locations.

5.6.2 Event Causality Guards

A set of guards are supplied for ensuring appropriate causality exists between events. There are three guard functions in this category, namely **before**, **occur**, and **replyto**. The **before** function accepts two event references and returns true if the first referenced event is causally before the second. The **occur** function takes an event reference and returns true if an event matching that reference exists in the causal history of the current event. The **replyto** function takes two event references and returns true if the second event referenced is causally preceded by the first. In addition the first event reference is restricted to events co-located with the guarded event.

The use of **before** and **occur** functions is fairly self-explanatory, however, the **replyto** guard deserves further discussion. This guard is used in contexts where a particular event can only be executed if the matching causal dependent (the second parameter) is generated as a result of a specific preceding event. This allows us, for example, to filter out stray replies to multicast requests that should be discarded. The

reply guard also serves to support an internal optimization in implementations: in a situation where there are multiple clients in an RPC interaction model, the default (and correct) behaviour would be to send the RPC reply to all clients even though only a single client made the request. This guard allows the server(s) to determine that only a single client can use the reply, therefore only one reply need be sent.

5.7 Parameter Relationships

An event template specification can include a specification of its parameter relationships. The specification defines the values of the parameters. For example:

```
e1!(x:t1, y:t2) -> e2?(z:t3) {z = f(e1.x)}
```

Parameter relationship specifications can refer to any identifiable, causally preceding event using the event reference semantics defined in section 5.5. There is no requirement that all parameters of any output event must be consumed by an input event, and the parameters of an output event can be used many times. Parameter relationships are functional, allowing for transformation of data. For all parameter relationships, the function or operator used must be well-defined for the data types of the parameters. This means, for example, that equality (=) can be used for parameters of different types provided it is well defined in the context of the binding. Due to its common use in remote procedure call, *Finesse* has shorthand syntax for name equivalence of parameters, that is:

```
e1!(x:t1, y:t2) -> e2?(x:t1, y:t2) {*= e1}
```

This specifies all parameters of `e2` with names matching parameters in `e1` are assigned the value of that same-named parameter. There is no requirement that all parameters in either `e1` or `e2` be assigned by the operator. Note that as described in the discussion of event reference semantics, the keyword **prev** can be used to refer to the immediately preceding event in the current specification context.

5.8 Reuse and Generics

The **Import** keyword allows role and binding definitions to be re-used in the current *Finesse* program. It is followed by the name of a *Finesse* program to import. The role and interaction definitions contained in that program are then able to be used within the current program. For roles, they can be referred to using `ProgramName.RoleName` syntax in a role definition only. This syntax simply copies the identified role definition from the imported program into the current program. If the role definition is to be used multiple times, a name scope must be defined to distinguish the instances.

The interaction definition contained within the program can be reused only within the interaction specification. It can be referenced using only the program name, but must be parameterized by a set of named role behaviours that match the roles of the imported binding. The semantics of this reference is to copy the interaction behaviour from the referenced program with the role name in any `Role.event` specification replaced by the fully-qualified behaviour name (i.e. `Localrole.behav.event`.

In the simple case, role and binding definitions are re-used without parameterization, for example:

```
Binding Message {
  Roles {
    Sender {send!(x:t1)}
    Receiver {receive?(x:t1)}
  }
  Interactions {
    Sender.send -> Receiver.receive {*=Sender.send}
  }
}


Binding UseMessage {
  Import Message;
  Roles {
    Send2 {send1 {Message.Sender}
           -> send2 {Message.Sender}}
    Recv2 {recv1 {Message.Receiver}
           -> recv2 {Message.Receiver}}
  }
  Interactions {
    Message(send1, recv1) AND
    Message(send2, recv2)
  }
}
```

The roles of the *Message* binding are used to define two actions each in the Send2 and Recv2 roles respectively. The interactions section of the *UseMessage* binding simply binds those actions together using the *Message* binding. While this is useful, the ability to parameterize roles with arbitrary parameter lists give more flexibility. This requires an incomplete binding program definition with placeholders for parameter lists, used as follows:

```
Binding Message {
  Roles {
    Sender (MSG) {send!(MSG)}
    Receiver (MSG) {receive?(MSG)}
  }
  Interactions {
    Sender.send -> Receiver.receive {*=Sender.send}
  }
}


Binding UseMessage {
  Import Message;
  Roles {
    Send2 {send1 {Message.Sender(x:t1)}
           -> send2 {Message.Sender(y:t2)} }
    Recv2 {recv1 {Message.Receiver(x:t1)}
           -> recv2 {Message.Receiver(y:t2)} }
  }
  Interactions {
    Message(send1, recv1) AND
    Message(send2, recv2)
  }
}
```

Wherever a placeholder name is encountered in the imported binding definition, it is replaced with the parameter list provided in the instantiation. This allows us to reuse the interaction behaviour with different event parameter lists, allowing definition of bindings such as generic RPC or multicast. Note in this case the usefulness of name-equivalence for parameter relationship specifications.

5.9 Concluding Remarks

This chapter has described the *Finesse* language, a coordination language for open distributed systems. A *Finesse* program or binding describes the roles of components in a

distributed application and the interactions between those roles. Roles and interaction are described using event relationships, in particular, causality (ordering), parameter and timing relationships. *Finesse* has strong support for group communication and provides abstraction through structuring and composition features.

Finesse has a number of features that are novel in coordination languages. Its key advantage is the ability to be executed over the distributed, asynchronous execution model defined in chapter 4. Also of particular interest is the abstraction that it provides over messaging. Messaging is implied by declarative relationships between events, meaning that a compiler or interpreter can optimize the number and content of messages transferred between components. The use of explicit, but abstract, parameter relationships allow parameters to be ignored if not used. The use of causality relationships allows parameters from multiple events to be combined into a single message from a particular interface where appropriate.

Openness and flexibility is enhanced by allowing arbitrary parameter relationships. This can allow, for example, a DCE RPC client to call a CORBA server, provided the appropriate infrastructure and transformation functions are in place. The *Finesse* language has no structural knowledge of data types, freeing it from the confines of a specific data model. The use of functional relationships between parameters also provides good support for including legacy components and applications in a *Finesse* binding.

The inclusion of time constraints is both novel and very useful. Such constraints can be used to explicitly specify timeouts and associated behaviour, or to describe quality of service constraints on, for example, the delivery of multimedia streams.

It should be emphasized that the *Finesse* language is an example of how the execution model described in the preceding chapter can be used. Other syntax definitions are possible and in particular, a graphical programming language might be particularly appropriate in a coordination context. The definition of such a language is not addressed by this thesis.

Chapter 6

Runtime Engine

6.1 Overview

This chapter describes the design of a distributed runtime engine implementing the *Finesse* semantics defined in chapter 4 and providing a run-time target for the *Finesse* language described in the previous chapter. A *Finesse* program is captured by a set of event templates that describe the possible partial orderings of events, their parameter relationships, and guard expressions for events which can include timing constraints. A correct execution of a program is a partial order of event execution that matches one of the partial orders described by the program, with each event satisfying the required parameter relationships and any guards.

The runtime engine executes a *Finesse* program by executing enabled events. The initial state of the program is represented by the set of events that have no predecessors, hence can be executed at any time provided their respective guards are satisfied. The execution of an event potentially enables other events which are subsequently executed and so on.

A *Finesse* program is executed by a set of distributed participants. Each distinct event is executed by a distinct participant in the program. The runtime engine hence operates as a set of distributed runtime engines, one for each participant. The participant engines maintain a graph of occurred and pending events. Pending events to be executed locally can be fired by a participant when all required predecessors are known to have occurred (it has sufficient offers), and when the event guard expres-

sion, if any, is satisfied. When an event is executed, a notification of that event is sent to those participants with potentially succeeding events, that is, events to which this event makes an *offer* of causality. The graph maintained by each participant is a partial view of the system state because remotely executed events might not be visible in the graph at any given time. In general, *no single participant will have a complete view of system state*. The participant runtime engine is thus responsible for maintaining the graph of occurred and pending events, executing local enabled events, and for delivering notifications of executed events to local software components and remote participants.

This design has been implemented in a prototype written in the Java language. The prototype does not fully implement the semantics described here, but is able to correctly execute most valid programs. The limitations of the prototype are noted in the appropriate sections.

Although not strictly required, the implemented semantics includes explicit support for some elements of the language. In particular, it uses the structuring concepts of *role* and *interface*, where a role defines co-located behaviour and an interface provides a specific location for role behaviour. The implementation allows multiple interfaces to fill a role, meaning that the event templates describing the role need to be distinguished by interface name in order to meet the semantic model requirement for an explicit association between an event template and a location.

The implementation also provides causal timestamp semantics for event notifications. This allows each participant to deterministically build a consistent causal history graph to resolve event references and enforce any ordering constraints implied by guards. While these semantics can be implemented at compile-time by other, more-specific mechanisms, our approach allows us to easily add additional guard semantics and also provides a more useful platform for debugging programs.

The following sections present the design of the runtime engine and its prototype, including the internal representation of programs, the management of enabling relationships, building a causal history graph, and a number of ancilliary issues.

6.2 Representing Programs

Each participant requires a copy of the program to be executed at initialization. This executable representation of the program describes the set of event templates that form the program. Each template is assigned to a role, and defines the predecessor requirements, successor requirements, guard expression, and parameter relationships expression. It also includes a set of forward parameter references, allowing the runtime engine to determine where each parameter value might be required. For internal processing efficiency, local successors are distinguished from remote successors.

The syntax of this internal form is a set of well-defined S-expressions. While not a high-level language, this form can be written by hand with minimal effort or could be the target of a *Finesse* language compiler. A parser for these expressions has been built to support the prototype implementation. The parser returns the set of event template objects as defined by the S-expressions, ready for execution by the runtime engine.

The association of interfaces and roles is performed at initialization time, so from the perspective of the semantic model, this association forms part of the program definition. The engine distinguishes between events templates executed at different interfaces having the same role at run-time. Since the *Finesse* language has been designed with multiply-filled roles in mind, this is not especially difficult and the implementation is discussed in a later section of the document.

6.3 Initialising a Program Execution

The execution semantics requires that all participants in a distributed execution begin with a copy of the program and an initially empty execution history. The *Finesse* language adds the ability to associate interfaces (locations) with roles (co-located behaviour) at initialization time. The approach taken to satisfy these requirements is as follows:

1. A *binder* is instantiated with a *Finesse* program in the internal form. It parses the role definitions and waits for offers to fill roles from participants.

2. Each participant engine is instantiated with the name of a role that they are able

to implement and the location of the binder. The participant engine advises the binder that it is able to fill the named role and nominates a communications endpoint (in this case, a TCP address) for participation in the program.

3. When the binder has sufficient participant offers to instantiate the binding (governed by the role cardinality constraints), it distributes the program and a list of participants to each participant and waits for an acceptance of that program from them.

4. Once the participants have accepted the program, a confirmation is sent to each of them and the program can begin.

There are a number of important points to note about this behaviour:

- The binder is considered to be a distinct entity, therefore allowing program instantiation by either one of the participants or a distinct controlling entity.

- Each participant advertises only their ability to fill a named role. They do not nominate a specific program. This allows, for example, a participant with an RPC client interface to operate unmodified in both a unicast and multicast RPC binding.

- It is quite possible for the advertising and acceptance of a program offer by a participant to be made somewhat separate from the execution of the program, for example, the advertising and acceptance of a program might be performed by a secured management application that checks the credentials of the participants prior to establishing the binding.

These features allow the implementation of functional behaviour to be entirely separated from the administrative tasks associated with creating a program instance. This is of particular interest for cross-enterprise interaction, for example electronic-commerce applications. If you consider the program to reflect the contract between the parties fulfilling the roles, it is quite likely that the contract negotiation will be complex and often different for each program instance. The separation of concerns provided by the initialization approach makes this quite feasible to implement. You could also, for

example, have distinct binding processes for internal and external interactions without modifying the functional behaviour.

6.4 Executing a Program

Each participant engine has a copy of the program currently being executed and maintains a graph representing the current system state, as seen by that participant. The graph maintained by each participant is built by adding events created from event templates to a pending portion of the graph when any of their immediate predecessors are known to have occurred or immediately if they have no direct predecessors. This portion of the graph is known as the *front*. The information about the ordering of events required for this addition is contained in the templates. When all predecessors are known to have occurred and the guard is satisfied, the event can be executed by the engine. When executed, the event is moved from the pending portion of the graph to the *history* which contains information about executed events. Note that an engine can only execute events associated with its role.

The graph therefore contains both events that have occurred and events that are expected to occur. Whenever a local event is fired or knowledge of a remote event occurrence is received, the event is placed in the history and any immediately succeeding events are added to the front if not already present. Each event in the front is then evaluated to determine if it can be executed.

When an event is executed by a participant, information associated with that event is transmitted to all other participants that have immediately succeeding events in the *Finesse* program, have parameter relationships with the executed event, or have guards that reference the executed event. This transmission carries a causal timestamp for the event, identifying recent events in the history of the sender and allowing other participants to correctly update their view of the system state. The execution of a *Finesse* program continues in this fashion until no further events can be enabled or an error occurs.

The following subsections outline the details of execution, including the enablement of events, evaluating guards, maintaining the history, and implementing parame-

ter relationships.

6.4.1 Causal Enablement

The causal enabling relationships between events can be quite complex because *Finesse* programs can specify a considerable level of non-determinism in the partial ordering of events. The successors and predecessors of an event can be grouped with logical AND, OR, or subset choice relationships as described in the preceding chapters. The subset choice relationship is equivalent to an OR with a cardinality constraint. These relationships can be nested to an arbitrary depth, so the runtime engine represents them as decision trees.

Each event (pending or not) requires distinct decision trees for predecessors and successors because the decision trees for enablement of events are not symmetric: the information contained in the successor decision tree of an event includes knowledge of all possibly succeeding events. Such information is not required or used by any individual successor. Similarly, the predecessor tree of an event includes knowledge of all possible predecessors and this information is not required or used by any individual predecessor. This separation corresponds to the distinct notions of the *offers* and *requires* sets in the semantic model.

A predecessor needs to evaluate its successor tree after each acceptance of an offer to determine if it can continue to offer its *causality* to other successors. Similarly, a pending successor needs to evaluate its predecessor tree to determine if it continues to require the causality of a predecessor. When possible causal relationships between predecessor and successor are broken, the edge can be removed from the graph. Note the implication that edges in the graph between events in the history and events in the front are tentative and only become fully fledged edges when both events are in the history (i.e. both have been executed).

6.4.2 Guards and Timing Relationships

The general form of a guard expression is a set of boolean-valued logical expressions connected by **AND**, **OR**, and **XOR** operators. These expressions can be nested. Expressions in guards can reference the parameters and timestamp of causally preceding

events. The general form of these expressions is a boolean valued Java method, or for numeric values, the usual comparison operators can be used. Event references are resolved using the causality information supplied in the causality vector described in section 6.4.4 below. Non-boolean values within expressions can be defined as literals (where supported), direct references to event parameters, or Java methods. Method parameters can be expressed using the above value expressions.

Once an event has sufficient offers to be causally enabled, it should always be possible to evaluate the guard and any event references that it contains, with one exception: short circuit evaluation can in some cases avoid resolving a parameter reference. In most cases, however, this should not be necessary.

Timing relationships in guards require special handling. There are explicit *TimeLess* and *TimeMore* guards that define a time relationship between the current event and a causally preceding event. The semantics of these have been defined in chapter 5. Any synchronization of time across distributed components is subject to an accuracy error, and should be taken into account when evaluating time constraints. The prototype implementation does not deal with this issue: it assumes that timestamps provided by an appropriate time synchronization algorithm (e.g. NTP) are sufficient for most purposes. Future versions, however, might attempt to determine communications latency and estimate a bound on the error. The error bound can then be applied to the edges of the time bands associated with each event.

6.4.3 Enabling an Event

The enablement of a pending event requires the satisfaction of both causal enablement requirements and the event guard. The guard, however, can only be evaluated correctly relative to a specific set of causally enabling events. We address this issue by through the pending event mechanism: a set of offers from predecessors is provisionally accepted by the pending event as knowledge of those predecessors is available. When the event is causally enabled, the guard is evaluated against the set of enabling events. If the guard fails, alternative offer sets are evaluated to see if any can enable the event.

The search space of offer sets is typically small but potentially quite large. When a guard fails, it is often because an alternative path has been chosen that does not include

the current event. There are a number of identifyable circumstances, however, where alternative offer sets can enable the guard:

- when multiple reply events are generated but only a single reply is used (e.g. the multicast RPC example), offers from the other replies will be passed to the next reply acceptance event. With a reply guard in place, these offers cannot be accepted and a more recent reply must be chosen.

- when a stream of events must be delivered in order but some events can correctly be dropped (e.g. an audio stream), an attempt could be made to deliver a dropped event out-of-order. An ordering guard would make the offer from this out-of-order event invalid.

- in an RPC binding with multiple instance of the client role, a reply guard might be used to ensure that a reply is only used by the requesting client instance. If replies are sent to all clients (the default behaviour associated with roles), the client must distinguish between replies to its own requests and other replies.

Note that in all cases the problem exists because there are multiple events with the same template name, either through iteration or multiply-implemented roles. While other circumstances can lead to alternative offer sets, we believe this is the most likely. We leave the efficient searching of this offer space to future research. Intelligent evaluation of guard logic should in most cases be able to minimize searching. The prototype implementation does not implement searching of the offer space at all. If the initially chosen set of offers does not satisfy the guard, the event becomes impossible. A number of heuristics are used to avoid some of the specific situations described above and the prototype thus allows correct programs to execute in most circumstances.

6.4.4 Managing the Event History

One of the implementation features is that the graph built by the runtime engine of each participant in a program execution properly reflects the causal relationships between events. The key issue is that we can only guarantee that a participant is aware of events it has executed locally, and any remote events can only be correctly positioned in the

graph if their relationship with local events is captured. Information about events executed by a remote participant therefore identify the nearest causally preceding events at the local participant.

To clarify this problem, consider the following example:

1. Participant P sequentially executes output events $A1$ and $A2$. These are transmitted to participant Q. P is expecting Q to execute an output event in reply to each of $A1$ and $A2$.

2. Participant Q executes events $B1$ and $C1$ in reply to $A1$ and $A2$ respectively.

3. Knowledge of $B1$ and $C1$ is received at P in a single message.

4. How can P determine how to match the requests and replies?

In the example above, the problem must be solved by having P supply an identifier that must be carried by any replies. Since we are already naming events to allow us to maintain a graph, it seems appropriate to use the event name. The situation is complicated by the fact that there can be an arbitrary number of intervening events executed by any of the participants, and the reply could generated by an event at a different remote participant that has received a subsequent request from Q. Request delegation is an example of this. A further complication is introduced when a remote event *joins* two distinct branches of a local computation: it must carry identifiers for both local events since they are causally independent.

Knowledge of an event is hence transmitted with a vector of event names identifying the nearest causal predecessors of each participant. By definition, an event executed from a particular template will causally succeed previous occurrences executed from that template. Since the resolution of event references is the most common use of this information, we include include the most recent occurrence of a causally preceding event from each event template at that participant. While this potentially increases the size of the vector, it allows simple and fast searches of the vector to resolve event references. When an event is executed, the identities of all events that causally enabled this event are included in the vector, along with the merger of their vectors. Any redundant entries in the new vector are removed (i.e. only the most recent of two same-named events need to be included).

The primary implication of the need to carry event identifiers is that the size of this event causality vector can become significant. It is important to note, however, that the size of the vector is bounded by the sum of the number of unique template names in each role multiplied by the cardinality of that role in a program instance. In most cases, the implicit and explicit parallelism in the program specification will ensure that the likely size remains well below this theoretical bound. This is particularly the case for multiply-implemented roles, since each instance of the role is executed in parallel with others. Note that our use of causality vectors is somewhat different from the general and unbounded case of vector clocks [40] because we have a bounded number of participants.

There are other opportunities for optimization. As mentioned in the introduction to this chapter, the need for causality vectors can be removed by intelligent compilation. Guard constraints that imply ordering can be replaced with explicit request identifiers or counters and guards that check those values. This avoids any overhead for programs and program fragments that do not require ordering constraints. It would also be possible to build an implementation of the semantic model that explicitly required all event references to be resolved by the set of offers made directly to the referring event. This model can implement the event reference semantics of the language by generating additional offer requirements at compilation-time, or the event reference semantics could be modified apply the restriction directly.

6.4.5 Parameter Relationships

A relatively simple approach to parameter relationships is employed in the implementation of *Finesse*. The data type associated with a parameter is defined as a string identifying the Java class of the data type. There are four possible ways of defining the value of a parameter:

- if left undefined, it is expected that the host program will provide a value for the parameter when initiating the event.
- if the data is a string, integer or real value, a literal value can be used
- by a direct reference to a parameter of a preceding event with the same type

- as a Java constructor or method that yields a value of the correct type. The parameters of this method can be expressed using literals, direct parameter references, or nested method calls.

Reference to preceding event parameters are resolved by searching the causality vector associated with the current event. By definition, the most recent causally preceding event of a given name will exist in the vector. A correct program cannot include parameter references to events that are not present in the vector because a parameter relationship must be supported by a causal relationship. Where multiple matching events from different participants exist, preference is given to a local event. Otherwise, the first located match is used.

6.5 Language Binding

The approach to language binding used in the implementation of the *Finesse* engine requires that the host program (that is, the program participating in the binding) synchronize with the engine on every event executed. This is a relatively low-level mechanism, but provides significant flexibility and power.

There are two calls made to the engine. The first call *readyFor* indicates that the host program is ready to execute a particular event, and provides any parameter values required for that event execution. An event execution handle is returned by the call. This call returns immediately without waiting for the event to occur. The second call *waitFor* accepts an array of event execution handles, and returns when any of the events indicated by those handles has been executed, or when any of the nominated events has become impossible through guard failure. An object containing the event handle and all event parameters is returned by this call.

Higher-level mechanisms can be easily built on top of this mechanism. It would be relatively simple to implement, for example, an object offering a number of remote procedure calls, each possibly having exceptions. The call from the host program would result in a *readyFor* call for the request event and any possible exception events, then a *waitFor* to ensure the call is made, followed by a *readyFor* call for the reply event and any possible exception events, then a *waitFor* to collect the reply. The ex-

ception events can be delivered through Java exception handling mechanisms if they occur, or the method could return normally when the reply event is executed. In many cases, such a wrapper could be automatically generated from the *Finesse* language program, but the lower-level mechanism is available for unusual cases that require special handling.

6.6 Other Design Issues

The following subsections discuss some additional design features that have not been implemented, but might be used in a more advanced implementation.

6.6.1 Garbage Collection

At any point in the execution, a participant will maintain a graph of events including both the history and the front. To avoid having the history grown to infinite size, it would be useful to implement a garbage collection policy. The rule for garbage collection can be easily defined as follows: events in the history can be garbage collected when no further offers from that event can be used by a local successor, and it is not possible for any local event to reference this event in a guard or parameter reference.

6.6.2 Error Detection and Handling

The prototype implementation does not implement error detection or handling. This section indicates, however, the types of errors that can be detected and how they should be handled. There are only two possible types of errors that can be detected by the runtime engine:

1. Incorrect behaviour of a participating component

2. The inability to continue execution due to a programming error

Incorrect behaviour of a component is a local issue and should not affect the other participants unless it leads to an inability to continue. An exception should therefore only be delivered to the local component. The local engine could, however, elect to

inform an auditing system if one existed. Detection of an inability to continue can be flagged at the local component and propagated to other components.

The runtime engine makes no assumption about communications reliability. When an event is executed locally, it sends a message containing the details of the event to those that require it, but does not in any way attempt to determine if the message arrives successfully or within a fixed time frame. If the *Finesse* program assumes that a particular communication will be successful, then this is a programming error and the engine will raise an exception if the communication is unsuccessful and it is detected.

This approach might seem draconian, however, it places the error handling responsibility squarely on the shoulders of the programmer. It is inherently dangerous to assume the reliability of communication in a distributed system, particularly those intended for Internet-scale networks. It is reasonable, however, to assume that communication will often be successful. Pre-packaged solutions that provide generic handling of communications failure can be coded as *Finesse* modules. This avoids the assumption that reliable communications is always desirable. In mobile computing, for example, a component is only occasionally connected. Using the CORBA request/timeout approach in such situations is undesirable. The *Finesse* engine thus provides flexibility so that the programmer can write their own communications error handling in situations that have special circumstances.

It is expected that an intelligent compiler for *Finesse* would analyse the code and pinpoint potential problems, including assumptions of communications reliability, possible deadlocks, and possible livelocks.

6.6.3 Reliable Communication

There is an interesting possibility for future research into making the engine automatically support reliable computing. We could add a condition that the executor of an event is responsible for ensuring that all remote participants that have immediate causal successors or that reference the parameters of the event are aware of the event. This implies that the event must be maintained at the executor until it receives acknowledgment that the event data has been received at all of these other participants. This acknowledgment could rely on causality vectors or piggy-backed acknowledgements

transmitted with subsequent events from those participants. Using this approach, there is potential for developing intelligent compilers or run-time engines that determine the most efficient acknowledgment strategy based on application semantics.

6.7 Concluding Remarks

This chapter has outlined the design of a distributed runtime engine to support the semantic model and language defined in the preceding chapters. The chapter presented the design of the runtime engine and its prototype, including the internal representation of programs, the management of enabling relationships, building a causal history graph, and a number of ancilliary issues.

The prototype implementation demonstrates the feasibility of the design and the underlying semantic model. While there are some efficiency and resource usage issues in the prototype that deserve further attention, the design is sound, matches the semantic model quite closely, and provides a solid basis for further research and investigation.

Chapter 7

End-to-End Example

This chapter captures the concepts of the preceding chapters in an end-to-end example of the Finesse system. We define a simple remote procedure call example and describe the compilation, instantiation, and execution of the program. The intent is to provide a high-level understanding of the way the pieces of the Finesse system fit together, reinforce understanding of the execution model, and establish a context for understanding more complex examples.

7.1 Example Program

The following example program describes a simple remote procedure call as a sequence of events: client send, server receive, server send, then client receive. A single parameter is sent with the request, and a single parameter is included in the reply.

```
Binding SimpleRPC {

  Roles {
    Client {
       send!(x:T1) -> receive?(y:T2)
    }
    Server {
       receive?(x:T1) -> send!(y:T2)
    }
  }

  Interactions {
     Client.send -> Server.receive {x = Client.send.x} AND
     Server.send -> Client.receive {y = Server.send.y}
  }
}
```

7.2 Compilation

As discussed in preceding sections, the compilation of the example results in a set of event templates that define the potential causal successor (*offers*) and predecessor (*requires*) relationships, any event guards, and any parameter relationships between events. We will describe the compilation in two passes with the first pass creating the base event templates from the `Role` specification and the second pass adding additional constraints and information from the `Interactions` specification.

For ease of understanding, the binding and event template data is presented in subsequent sections as a description list rather than the S-expression syntax used by the prototype implementation.

7.2.1 Roles Compilation

The roles compilation results in the capture of the role names, their cardinality and a set of base event templates:

Roles	Client, cardinality 1	
	Server, cardinality 1	
Client send	Role:	`Client`
	Event name:	`send`
	Guard:	`true`
	Parameters:	`(x:T1)`
	Requires:	`()`
	Offers:	`(Client.receive)`
Client receive	Role:	`Client`
	Event name:	`receive`
	Guard:	`true`
	Parameters:	`(y:T2)`
	Requires:	`(Client.send)`
	Offers:	`()`
Server receive	Role:	`Server`
	Event name:	`receive`
	Guard:	`true`
	Parameters:	`(x:T1)`
	Requires:	`()`
	Offers:	`(Server.send)`
Server send	Role:	`Server`
	Event name:	`send`
	Guard:	`true`
	Parameters:	`(y:T2)`
	Requires:	`(Server.receive)`
	Offers:	`()`

7.2.2 Interactions Compilation

The compilation of the interactions specification adds remote predecessor and successor relationships and the necessary parameter relationships. It can also potentially add guard expressions. For both guards and parameter relationships, the new terms are added with a logical `AND`.

Roles	Client, cardinality 1	
	Server, cardinality 1	
Client send	Role:	`Client`
	Event name:	`send`
	Guard:	`true`
	Parameters:	`(x:T1)`
	Requires:	`()`
	Offers:	`(Client.receive)`
Client receive	Role:	`Client`
	Event name:	`receive`
	Guard:	`true`
	Parameters:	`(y:T2) (y = Server.send.y)`
	Requires:	`(Client.send AND Server.send)`
	Offers:	`()`
Server receive	Role:	`Server`
	Event name:	`receive`
	Guard:	`true`
	Parameters:	`(x:T1) (x = Client.send.x)`
	Requires:	`(Client.send)`
	Offers:	`(Server.send)`
Server send	Role:	`Server`
	Event name:	`send`
	Guard:	`true`
	Parameters:	`(y:T2)`
	Requires:	`(Server.receive)`
	Offers:	`(Client.receive)`

7.3 Participant Behaviour

The compiled program describes the distributed execution behaviour associated with a simple remote procedure call. It is important to remember, however, that the program only defines the visible behaviour of the participant programs and not the internal behaviour of the participants themselves. In order to execute the program, we require two participants that have externally visible behaviour compatible with the Client and Server roles respectively. We do not prescribe any particular notion of compatibility: if a participant attempts to execute incorrect behaviour, the runtime engine will throw an exception back to the participant. As described in the preceding chapter, a Java API is defined for participants to interact with the runtime engine. This API allows the participant to synchronize with the engine on locally executed events, including the transfer of parameter values into and out of an instance of the Finesse engine.

In this case, the `Client` participant must synchronize with the engine on the `send` event and provide an appropriate value for the parameter `X`, then synchronize with the engine on the `receive` event to retrieve the `Y` parameter value returned by the remote procedure call. The `Server` participant must synchronize with the engine on the `receive` event and retrieve the `X` parameter value supplied by the client, then synchronize with the engine on the `send` event and provide an appropriate value for the `Y` parameter to return to the client.

7.4 Instantiation

On startup, each participant or a manager process for the participant creates an instance of the Finesse engine to execute the distributed program. The responsibility for instantiating a binding across the participant engines can be taken by any party, including both participants and third parties. The prototype environment uses a separate binder process that waits for sufficient offers from participants to fulfill the roles in a predefined binding. Every participant identifies a communications address for their engine. This process is arbitrary, however, and mechanics of binding creation are somewhat orthogonal to binding execution except that on instantiation, each engine must know the binding program being executed, the set of participants, and the role of each participant.

7.5 Execution

Once the binding has been instantiated, each engine parses the binding program and builds a set of event templates and two data structures: a *history* and a *front*. The history contains events that are known to have been executed and is initially empty. The front contains the set of events that are awaiting execution and is initially populated with those events with no mandatory causal predecessors.

The following descriptions describe the execution of the example binding program in terms of the enabling of events, their execution, and the subsequent notification of other participants.

7.5.1 Initial Client State

The client role involves templates for the `send` and `receive` events with the `send` event initially in the front and enabled, since it has no predecessors and a true guard. We depict the state graphically follows:

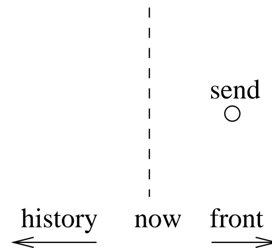


Figure 7.1: Initial state of client

7.5.2 Initial Server State

The server role also has two event templates for the `receive` and `send` events, but neither is initially in the front.

7.5.3 Client Executes Send

The only event initially enabled is the client `send` event and this event is executed as follows:

1. the engine synchronizes with the participant on the event, obtains a value for the `X` parameter, and binds other environment values (e.g. time) appropriately;

2. the send event from the front is moved to the history;

3. the local successor of the event (`receive`) is added to the front with an offer of causality from the `send` event;

4. the participant executing the `Server.receive` event is notified of the event occurrence.

The state of the client engine is now:

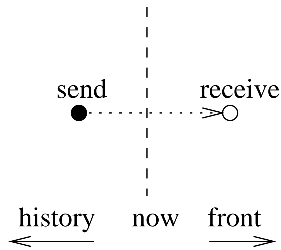


Figure 7.2: Client state after send

Note that the `receive` event in the front still requires a `Server.send` event before it becomes enabled.

7.5.4 Server Receives Client Send Notification

On receipt of the `Client.send` notification the following steps are performed in the server engine:

1. the `Client.send` event is added to the history;
2. the `Server.receive` successor event is created and added to the front with an offer of causality from that client event.

The state of the server engine is now:

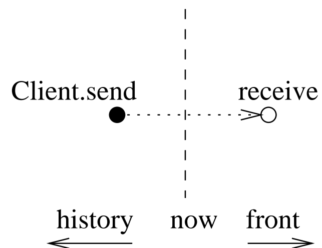


Figure 7.3: Server state after receiving client notification

7.5.5 Server Executes Receive

The server engine now has its `receive` event enabled and a value for the `X` parameter. The engine executes the following steps:

1. the engine synchronizes with the participant on the receive event with the `X` parameter and other environment values bound appropriately;
2. the receive event in the front is moved to the history;
3. the local succesor event (`send`) is added to the front with an offer of causality from the `receive` event.

The state of the server engine is now:

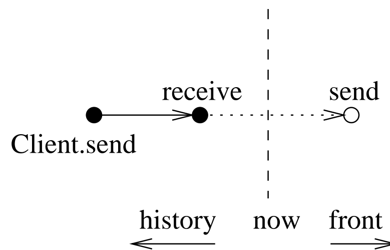


Figure 7.4: Server state after receive

7.5.6 Server Executes Send

After performing appropriate application-specific processing the server participant generates a value for the return value `Y` and the following steps are performed in the engine:

1. the engine synchronizes with the participant on the send event, obtains a value for the `Y` parameter, and binds other environment values appropriately;
2. the send event in the front is moved to the history;
3. the participant executing the `Client.receive` event is notified of the event occurrence.

The state of the server engine is now:

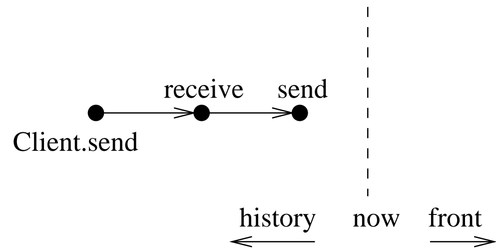


Figure 7.5: Server state after send

The server engine can determine from the binding program that no further events are possible at the server participant and it can therefore terminate.

7.5.7 Client Receives Server Send Notification

On receipt of the `Server.send` event notification, the client engine performs the following steps:

1. the `Server.send` event is added to the history;

2. the causal relationship between the `Client.send` and `Server.send` events is recorded, noting that as discussed in the preceding chapter, causal relationship information is transferred with event notifications in the prototype;

3. an appropriate causality offer from the `Server.send` event is added to the `receive` event in the front.

The state of the client engine is now:

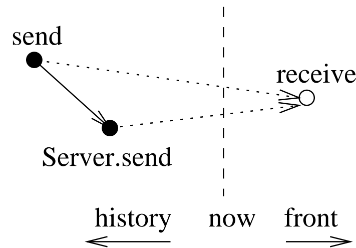


Figure 7.6: Client state after receiving server notification

7.5.8 Client Executes Receive

The receipt of the notification has added the causality offer required for execution of the `receive` event at the client participant. The client engine thus performs the following steps:

1. the engine synchronizes with the participant on the receive event with the `Y` parameter and other environment values bound appropriately;
2. the `receive` event is moved from the front to the history.

The state of the client engine is now:

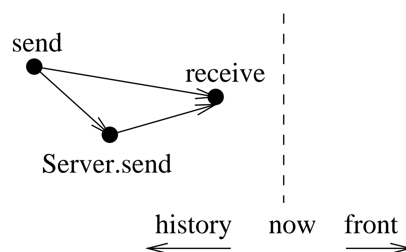


Figure 7.7: Client state after receive

The client engine can determine from the binding program that no further events are possible at the server participant and it can therefore terminate.

7.6 Concluding Remarks

This chapter has described the compilation, instantiation, and execution of a simple remote procedure call programming example in the Finesse system. In working through this end-to-end example, we have shown how the pieces of the Finesse system work together and described the execution flow of a simple program. While the example presented excludes some of the features of the language, execution model, and runtime engine, the example establishes a context for understanding more complex programming examples.

Chapter 8

Examples

This chapter uses a set of examples to showcase the power and flexibility of the programming model and language described in the preceding chapters. The examples address four distinct applications of *Finesse*:

1. Programmable, component-oriented middleware

2. Enterprise application integration (EAI)

3. Business to business interaction (B2B)

4. Computer supported cooperative work (CSCW)

Through the variety in the examples presented, we will demonstrate the flexibility, utility, and power of the *Finesse* approach. This variety also suggests that the semantics and structure used by the *Finesse* approach is suitable for a general purpose distributed software platform, or in other words, an appropriate "virtual machine" for execution for distributed software.

The examples are presented in the *Finesse* language syntax with commentary to describe the example and its significance. The mapping from the *Finesse* syntax onto the semantic model and thus the runtime engine can be inferred from the language definition presented in chapter 5. All examples have been parsed by a prototype language parser using the grammar defined in appendix A.

8.1 Programmable, Component-Oriented Middleware

One of the key advantages of *Finesse* is the ability to "program" the middleware. This programmability and the import semantics of the *Finesse* language allows the production of modular and re-usable interaction components. The examples presented in the overview of chapter 2 have already hinted at the possibilities for a programmable distributed systems middleware. In this section, we define a set of additional examples that complement the RPC examples and further demonstrate the utility of the approach.

The key point in these examples is that the middleware platform can provide both a runtime engine using the *Finesse* execution model and a set of pre-programmed library modules to implement common distributed systems interaction protocols. A programmer can choose to use the provided modules or implement a custom solution where those modules do not meet the application requirements. The modules can encapsulate proven solutions, thus minimising the likelyhood of lower-level protocol errors and allowing the application developer to focus on the application-level issues.

8.1.1 Reliable Multicast

Multicast protocols are useful for process group abstractions and implementing highly-available services. While the Finesse language has a convenient abstraction for multicast through roles that can be filled by multiple participants, this does not provide any guarantees of delivery. We have two choices in providing reliable multicast: build a runtime engine on a messaging infrastructure that guarantees delivery, or provide a module for reliable multicast. The former is likely to be inefficient in many situations, thus we demonstrate the latter in the following example.

```
Binding RelMulticast {
  Roles {
     [#=1] Sender(MSG, TIMEOUT) {
       send!(MSG) ->
       loop {
         [occur(resend) AND timeless(resend, TIMEOUT)
          XOR NOT(occur(resend) AND timeless(send, TIMEOUT)) ]
         resend!(MSG)
       } -> done?
    }
    [#>=1] Receiver(MSG) {
      receive?(MSG) -> loop { ack! }
    }
  }
  Interactions {
    Sender.send -> [#<=all] Receiver.receive {*= prev}
    AND Sender.resend ->
      [#<=all AND NOT occur(Receiver.receive)]
        Receiver.receive {*= prev}
    AND Sender.resend ->
      [#<=all AND NOT replyto(Receiver.ack, Sender.resend)]
        Receiver.ack
    AND [#=all] Receiver.ack -> Sender.done
  }
}
```

Note the use of the `replyto` guard: this guard is true if an event referenced by the first parameter is causally before the event referenced by the second parameter. In this case, this guard specifies that the engine need only resend to those receivers whose acknowledgements were not visible at the sender when resending.

While this example presents a simple reliable multicast with potentially unbounded retries, it is relatively straightforward to extend the example for a more complex implementation. For example, a more complex timeout could be used with a exponentially increasing timeout up to some configured limit.

8.1.2 Two-phase Commit

The availability of reliable multicast makes the specification of a two-phase commit protocol somewhat easier. We demonstrate this in the following specification:

```
Binding TWOPC {
  Import RelMulticast;
  Roles {
    Manager {
      begin { RelMulticast.Sender((), 5) } ->
      prepare { RelMulticast.Sender((), 5) } ->
      commit { RelMulticast.Sender((), 5) }
      XOR [timesince(prepare) > 60]
          rollback { RelMulticast.Sender((), 5) }
    }
    Resource {
      begin { RelMulticast.Receiver(()) } ->
      prepare { RelMulticast.Receiver(()) } AND
      prepare.receive ->
      loop { [occur(ready) AND timesince(ready) > 5
              XOR NOT occur(ready)] ready!() } ->
        commit { RelMulticast.Receiver(()) }
        XOR rollback { RelMulticast.Receiver(()) }
      XOR loop { [occur(abort) AND timesince(abort) > 5
                  XOR NOT occur(abort)] abort!() }
        XOR rollback { RelMulticast.Receiver(()) }
    }
  }
  Interactions {
    RelMulticast(Manager.begin, Resource.begin) AND
    RelMulticast(Manager.prepare, Resource.prepare) AND
    RelMulticast(Manager.commit, Resource.commit) AND
    RelMulticast(Manager.rollback, Resource.rollback) AND
    [#=all] Resource.ready ->
      RelMulticast(Manager.commit, Resource.commit)
    XOR [#>=1] Resource.abort ->
      RelMulticast(Manager.rollback, Resource.rollback)
  }
}
```

It is useful to note that an intelligent compiler could determine that the acknowledgements required for the prepare multicast can be piggybacked with notifications of the `ready` or `abort` actions of the resource role. We could remove the acknowledgements of the `prepare` event by explicitly specifying a reliable multicast `prepare` RPC with either a ready or abort outcome, but the current specification is somewhat simpler.

Note that this example only describes the interactions between manager and resources: it does not address the necessary behaviour of the manager and the resources to guar-

antee the protocol, that is, the need for resources to hold any locks from the time a ready response is returned to the manager until the transaction is either committed or aborted, and the need for the transaction manager to write the decision to commit or abort to stable storage and allow the decision to be recovered if required. The Finesse language is not intended to address these issues.

Remember that this two-phase commit program is a module that can be used in other programs, with the behaviour being synchronized with application behaviour through appropriate specifications. For example:

```
Binding StoreX {
  Import TWOPC;
  Import RelMulticast;
  Roles {
    Client {
      mgr { TWOPC.Manager } AND
      mgr.begin
      -> store { RelMulticast.Sender((x:Data)) }
      -> mgr.prepare
    }
    [#>=1] Server {
      res {TWOPC.Resource} AND
      store { RelMulticast.Receiver((x:Data)) }
    }
  }
  Interactions {
    TWOPC(Client.mgr, Server.res) AND
    RelMulticast(Client.store, Server.store)
  }
}
```

The only necessary synchronization is to execute the transaction manager begin multicast before storing, then execute the transaction manager prepare and by implication, the commit protocol, after storing.

8.1.3 Streaming Data

The transmission of streaming data is becoming a common form of interaction between distributed participants. In this subsection, we first describe a basic streaming data interaction and its use in an example. We then extend the module to provide lossy

streaming data that could be used, for example, in a streaming audio application where occasional packet loss is acceptable. We similarly extend the base streaming data module to define a data stream with minimum latency constraints.

In all cases, the endpoint role definitions remain unchanged. The example thus demonstrates the ability of *Finesse* to describe a variety of interation protocols including quality of service constraints, and the flexibility that comes from clearly distinguishing role and interaction behaviour.

We begin by defining the basic streaming data interaction:

```
Binding BasicStream {
  Roles {
    [#=1] Producer (DATA) {
      repeat {
        [occur(produce)] produce!(seqnr, DATA)
           {seqnr = add(prev.seqnr,1) }
        XOR [NOT occur(produce)] produce!(seqnr, DATA)
           {seqnr = 0}}
        -> stop!
    }
    [#>=1] Consumer (DATA) {
      while [NOT occur(stop)] {
        consume?(seqnr, DATA)
        XOR stop?
      }
    }
  }
  Interactions {
    Producer.produce ->
      { [#=all AND occur(consume) AND
           produce.seqnr = add(consume.seqnr,1)]
        Consumer.consume {*= prev}
        XOR [#=all AND NOT occur(consume)
           AND produce.seqnr = 0]
        Consumer.consume {*=prev}} AND
    Producer.stop -> [#=all] Consumer.stop
  }
}
```

This defines an interaction with a single producer and one or more consumers. The roles in this binding are parameterized by a list of data items included in each data packet. Remember from section 5.7 that the “`*= prev`” parameter relationship

statement indicates that all parameters of the event are assigned from same-named parameters in the preceding event. An explicit sequence number parameter is introduced to specify that packets must be delivered in order. Note that the `consume.seqnr` reference in the guard on the `Consumer.consume` event refers to the causally preceding occurrence of the event. The producer signals the end of the data stream by executing a stop event.

This module could be used, for example, to describe a video broadcast with both audio and video streams:

```
Binding BroadcastVideo {
  Import BasicStream;

  Roles {
    [#=1] Broadcaster {
      video {BasicStream.producer((vidframe:Picture))} AND
      audio {BasicStream.producer((audFrame:SoundByte))} AND
      video.stop -> [timeless(prev,0.1)] audio.stop
    }
    [#>=1] Receiver {
      video {BasicStream.consumer((vidframe:Picture))} AND
      audio {BasicStream.consumer((audFrame:SoundByte))}
    }
  }
  Interactions {
    BasicStream(Broadcaster.video, Receiver.video) AND
    BasicStream(Broadcaster.audio, Receiver.audio)
  }
}
```

The use of the stream module in this example is relatively straightforward, with the exception of the stop interaction. We have added a requirement that the broadcaster audio `stop` event occurs within 0.1 seconds of the video `stop` event. Note, however, that we haven't specified any other synchronization requirements across the streams.

We now extend the initial stream to allow the loss of packets:

```
Binding LossyStream {
  Roles {
    [#=1] Producer (DATA) {
       BasicStream.Producer(DATA)
    }
    [#>=1] Consumer (DATA) {
       BasicStream.Consumer(DATA)
    }
  }
  Interactions {
    Producer.produce ->
      { [#=all AND occur(consume) AND
            subtract(produce.seqnr, consume.seqnr) <= 2]
        Consumer.consume {*= prev}
        XOR [#=all AND NOT occur(consume)
             AND produce.seqnr <= 1]
        Consumer.consume {*= prev}} AND
    Producer.stop -> [#=all] Consumer.stop
  }
}
```

This is a lossy packet stream allowing at most one in every two packets to be lost. Note that a more complex specification of packet loss constraints could be implemented by introducing a `lostpacket` event that records the sequence number of the last lost packet and placing constraints on the “distance” between lost packets. While it is clear that such an event need not involve the consuming component, it requires a modification to the consumer role because the event becomes part of that role. A future version of the language could allow for anonymous events in a role that do not require participation from the component. Such a `lostpacket` event could be implemented using this mechanism and thus make the role compatible with the `BasicStream.Consumer` role. While the underlying semantic model is unchanged, this would also require a modification to the engine implementation since it currently requires synchronization with the component on all events.

A further extension of the stream interaction could impose latency constraints on the delivery of packets:

```
Binding FastStream {
  Import SimpleStream;

  Roles {
    [#=1] Producer (DATA) {
       SimpleStream.Producer(DATA)
    }
    [#>=1] Consumer (DATA) {
       SimpleStream.Consumer(DATA)
    }
  }
  Interactions {
    SimpleStream(Producer, Consumer) AND
    Producer.produce ->
      [#=all AND timeless(prev,0.1)] Consumer.consume
  }
}
```

This adds a timing constraint to the relationship between `produce` and `consume` events, requiring that the consumption events occur within 0.1 seconds of the produce event. Note that the execution of this specification requires closely synchronized clocks since the produce and consume events occur at different locations. The `AND` of the two behaviours in the `Interactions` section forces synchronization on same-named events and the appropriate merging of their guard expressions. Note that a more complex interaction specification could include alternative behaviour if the latency constraint is not met. A similar extension could be applied to the previously defined lossy stream:

```
Binding LossyFastStream {
  Import LossyStream;
  Roles {
    [#=1] Producer (DATA) {
       LossyStream.Producer(DATA)
    }
    [#>=1] Consumer (DATA) {
       LossyStream.Consumer(DATA)
    }
  }
```

```
  Interactions {
    LossyStream(Producer, Consumer) AND
    Producer.produce ->
      [#=all AND timeless(prev,0.1)] Consumer.consume
  }
}
```

It is easy to imagine more complex behaviours that define bandwidth constraints and other quality of service attributes on the interaction. Note, however, that the run-time engine can only enforce those constraints: there is no mechanism in place to choose a communication medium at initialization time that ensures these constraints will be met.

As suggested in the section introduction, these streaming data examples demonstrate the use of the language for building modules to support non-trivial interaction protocols involving quality of service constraints. The fact that all of the above streaming data modules use compatible role specifications shows that the use of configurable interaction mechanisms to connect relatively static components is both feasible and useful. We do note the restriction on introducing new events for control purposes, however, this is syntax and implementation issue rather than a fundamental problem in the semantic model.

8.2 Enterprise Application Integration

The use of enterprise application integration (EAI) toolkits to define and manage the integration of enterprise-level components is becoming prevalent. There are a number of such toolkits in the commercial software marketplace, with large vendors like IBM and Tibco having a significant presence. While these toolkits allow the definition of integration in a similar manner to the *Finesse* language, they all require the introduction of a centralized broker process to manage each integration. It is clear that this centralized broker becomes both a single point of failure and a performance bottleneck. *Finesse* allows the centralized broker to be removed and replaced with a set of asyncronous distributed runtime engines that are co-located with the components. We thus remove the bottleneck and single point of failure.

The following example demonstrates the ability of *Finesse* to replace an EAI toolkit in the integration process. We use the process of creating a new customer for a telecommunications provider as our example problem. The process typically involves a front-end application implementing customer management to interact with the customer (either directly through the Internet or through a customer service representative), with the customer first having a credit check, then appropriate details being forwarded to a billing system and a provisioning system. As a final step, the customer management system is notified when the process is complete so that a welcoming letter can be sent to the customer.

```
Binding NewCustomer {
  Roles {
    CustMgr {
       newCust!(data:CustOrder) -> welcomeLetter!(letter:PDF)
    }
    Provisioning {
       newService?(data:ServiceSpec) -> ready!
    }
    Biller {
       newAccount?(data:AccountSpec) -> ready!
       -> activate?
    }
    Credit {
       checkCredit?(person:PersonSpec) -> creditOK!
    }
    PrintService {
       print?(letter:Postscript, address:Address)
    }
  }
  Interactions {
    CustMgr.newCust ->
      Credit.checkCredit {person = personOf(newCust.data)}
    AND Credit.creditOK ->
      Biller.newAccount {data = accountOf(newCust.data)}
    AND Credit.creditOK ->
      Provisioning.newService {data = serviceOf(newCust.data)}
    AND Biller.ready -> CustMgr.welcomeLetter
    AND Provisioning.ready -> CustMgr.welcomeLetter
    AND Provisioning.ready -> Billing.activate
    AND CustMgr.welcomeLetter ->
      PrintService.print {letter=PDF2PostScript(prev.letter),
                          address=adddressOf(newCust.data) }
  }
}
```

8.3 Business to Business Interaction

A key process in almost all businesses is the purchasing of goods, and the following example describes a purchasing process to be implemented by *Finesse*. The importance of B2B interaction is reflected by the fact that a number of existing research prototypes and products already address the problem. This existing research and the advantages of using *Finesse* are discussed in chapter 9.

The purchasing process described by this binding involves a purchaser, supplier, a transport organization, and a bank. The process is begun with the issuing of a purchase order, and concludes with the payment for the order through the bank. We will assume in this binding that the purchaser and supplier operate in different countries, so an exchange rate calculation is necessary. The program described explicitly captures the detail of when a purchase is considered to be delivered, when the payment is expected, and the date used for calculation of the exchange rate.

```
Binding Purchase {
  Roles {
    Purchaser {
      locale!(locale:Locale) ->
        purchase!(order:Order) -> quote?(amount:Real) ->
          { accept!() -> pay!()
            XOR reject!() }
    }
    Supplier {
      locale!(locale:Locale) ->
        request?(order:Order) -> quote!(amount:Real) ->
        { accepted?() -> ready!()
          XOR rejected?() }

    }



    Banker {
      loop {
        exchange?(currA:Currency, currB:Currency, date) ->
          rate!(rate:Real)
      } AND
      accepted?() -> payment?(amount:Real)
      XOR rejected?()
    }
```

```
  Freighter {
    accepted?() ->
      pickup?(from:Address, to:Address) ->
        delivery!(date:Date)
    XOR rejected?()
  }
}

Interactions {
   -- establish a basic exchange rate for quoting
   {Purchaser.locale AND Supplier.locale} ->
     Banker.exchange{currA=currencyOf(Purchaser.locale),
                     currB=currencyOf(Supplier.locale),
                     date=today} AND

   -- purchase order and quote
   Purchaser.purchase ->
     Supplier.request {order = prev.order} AND
   {Supplier.quote AND Banker.rate}->
     Purchaser.quote
       {amount =
        div(Supplier.quote.amount.Banker.rate.rate)} AND

   -- quote accepted or rejected
   Purchaser.accept -> {Supplier.accepted AND
     Freighter.accepted AND Banker.accepted} AND
   Purchaser.reject -> {Supplier.rejected AND
     Freighter.rejected AND Banker.rejected} AND

   -- deliver and pay
   Supplier.ready ->
     Freighter.pickup {from=Supplier.locale.address,
                       to=Purchaser.locale.address} AND
   Freighter.delivery ->
     {Banker.exchange {currA=currencyOf(Purchaser.locale),
                       currB=currencyOf(Supplier.locale),
                       date=prev.date} AND
      Purchaser.pay} AND

   -- purchaser accepts risk of exchange rate
   -- variation since quote
   Purchaser.pay ->
     Banker.payment
       {amount =
        div(Supplier.quote.amount,Banker.rate.rate)}
 }
}
```

There is an additional point to note in this example and the previous example of

EAI interaction. In both examples, the high level interactions are described with little or no allowance for communications problems, and in particular, the loss of notifications. The existing prototype makes no guarantees of communications reliability, and would fail to execute the binding (through unbounded blocking) if any messages carrying event notifications are lost. This is not an intrinsic problem with the model or the implementation: we can easily address the problem by replacing the communications layer in the prototype with an alternative mechanism that guarantees reliability, for example, transactional messaging. Products supporting transactional messaging are readily available and would be the obvious target for an implementation of the model and language aimed at supporting EAI or B2B interactions.

8.4 Computer Supported Cooperative Work

One of the early motivating influences in this work was the need to provide a flexible platform upon which to build computer supported cooperative work (CSCW) environments. The key criticism aimed at existing middleware platforms was the lack of flexibility in interaction models, since CSCW applications often require a combination of highly synchronous interaction, streaming data, loosely replicated services, and asynchronous interaction. In the following example, we describe a CSCW "workspace" for a teaching scenario using such a varied set of interaction mechanisms. While we re-use our prior examples as much as possible, we also assume the existence of a number of other modules supporting additional interaction types.

The classroom example has a teacher, a mediator, students with video capability, and students with only audio capability. The class interaction consists of an audio and video broadcast, a slide presentation, a whiteboard for ad-hoc descriptions, and a text "chat" channel for student questions. The SlideShow and WhiteBoard bindings have an explicit mediator role to allow a pointer to be "handed" to students when questions are being discussed. We expect that the mediator would be a third-party who manages the interaction between students and the teacher, however, there is no reason why the person taking on the `Teacher` role could not also take on the `Mediator` role.

```
Binding Classroom {
  Import BroadcastVideo;
  Import SlideShow;
  Import WhiteBoard;
  Import Chat;
  Import Termination;

  Roles {
    Teacher {
      { video { BroadcastVideo.Broadcaster } AND
        slides { SlideShow.Presenter } AND
        whiteboard { WhiteBoard.User } AND
        chat { Chat.Reader } }
        -> term { Termination.Initiator }
    }
    Mediator {
      { slides { SlideShow.Mediator } AND
        whiteboard { WhiteBoard.Mediator } AND
        chat { Chat.Reader } }
        -> term { Termination.Participant }
    }
    [#>=0] VidStudents {
      { video { BroadcastVideo.Receiver } AND
        slides { SlideShow.Watcher } AND
        whiteboard { WhiteBoard.User } AND
        chat { Chat.ReaderWriter } }
        -> term { Termination.Participant }
    }
    [#>=0] AudStudents {
       { audio { BroadcastVideo.Receiver.audio }AND
        slides { SlideShow.Watcher } AND
        whiteboard { WhiteBoard.User } AND
        chat { Chat.ReaderWriter } }
        -> term { Termination.Participant }
    }
  }
```

```
  Interactions {
     BroadCastVideo (Teacher, VidStudents) AND
     BasicStream (Teacher.video.audio,
                  AudStudents.audio) AND
     WhiteBoard (Mediator.whiteboard,
                 (Teacher.whiteboard,
                  VidStudents.whiteboard,
                  AudStudents.whiteboard)) AND
     SlideShow (Teacher.slides, Mediator.slides,
                (VidStudents.slides,
                 AudStudents.slides)) AND
     Chat ((VidStudents.chat, AudStudents.chat),
           (Teacher.chat, Mediator.chat)) AND
     Termination (Teacher.term,
                  (Mediator.term, VidStudents.term,
                   AudStudents.term))
  }
}
```

This example highlights a number of interesting features of the language:

- subordinate modules can integrate common behaviour of distinct roles, for example, the teacher and students are peers in the `WhiteBoard` interaction.
- the conjunction of behaviours implies that same-named behaviours in distinct operands of the conjunction are the same behaviour. This allows a behaviour to be bound in more than one interaction specification.
- the audio-only students are bound to the audio component of the video broadcast: the ability to reference component behaviours of roles makes a separate audio-only behaviour for the teacher unnecessary.

While this example has not explicitly specified the detail of the SlideShow, WhiteBoard and Chat interactions, it clearly demonstrates how a framework for building CSCW programs could be provided in a consistent fashion using the *Finesse* platform. From this platform it should be relatively straightforward, for example, to provide a visual programming environment based on a set of common CSCW *Finesse* modules for interconnecting users and a set of GUI widgets providing user interfaces for the roles in those modules.

8.5 Concluding Remarks

This chapter has presented a set of examples to showcase the power and flexibility of the programming model and language described in the preceding chapters. The examples address four distinct applications of *Finesse*:

1. Programmable, component-oriented middleware
2. Enterprise application integration (EAI)
3. Business to business interaction (B2B)
4. Computer supported cooperative work (CSCW)

The wide variety in the examples presented shows the flexibility, utility and power of the *Finesse* approach. This variety also demonstrates that the semantics and structure used by the *Finesse* approach is suitable for a general purpose distributed software platform. It is worthwhile to emphasize that all of the programs can be executed by an aynchronous, distributed, runtime engine with no centralized view of the program state. The only synchronization is that explicitly specified in the programs. This is a significant feature of the approach that distinguishes it from existing distributed programming environments. This combination of capabilities suggests that the *Finesse* semantics defines an appropriate "virtual machine" for execution for distributed software.

Chapter 9

Discussion and Related Work

This chapter discusses the *Finesse* platform presented through specification and examples in the preceding chapters. In particular, it describes how *Finesse* addresses the assertions of the thesis introduction, highlights the strengths and weaknesses of the approach and implementation, and compares *Finesse* with related work from a number of research disciplines. We conclude that *Finesse* provides a number of unique advantages in the construction of distributed systems.

9.1 Does it Satisfy?

The introduction to this thesis made three assertions about the needs of distributed software construction and claimed that the *Finesse* approach presented in this thesis addresses those assertions. Having now described *Finesse* in detail, we return to those assertions and show that they are addressed.

The first assertion stated that:

> *The construction of software has become an evolutionary rather than revolutionary process. New software must extend or incorporate old software.*

The key feature of *Finesse* that addresses this requirement is the separation of interface (role) and interaction. The interface definition specifies how a component expects to interact with its environment without tying that component to explicit external com-

ponents. An existing software component can be integrated with new components by providing an interface defining the behaviour that is to be exposed. The interaction specification can then define the relationship between the components. The individual components have no need to explicitly reference other components: this is done by the interaction specification, which can be replaced or refined whenever new components are added and/or old components removed. It is also feasible, for example, to support migration from an old implementation to a new one by including both components in the integrated system until the new implementation is proven.

This approach has been promoted by research in coordination and architecture description languages, and its effectiveness is evidenced by the recent uptake of EAI toolkits in industry. As discussed both in preceding chapters and in subsequent sections, *Finesse* enhances the existing approaches through its powerful and flexible declarative description of behaviour, and through the ability to fully distribute the interaction behaviour.

The second assertion stated that:

> *Distributed applications must coordinate the activities of multiple participants with varying relationships. Static abstractions like client-server are too primitive and inflexible to describe such relationships.*

The examples presented in the previous chapter demonstrate the ability of *Finesse* to support multiple participants and varying relationships between participants. It is worth emphasizing, however, that the majority of current middleware platforms provide a small set of static interaction models: typically, they provide some combination of remote procedure call, messaging, and publish/subscribe interaction. Some platforms also provide stream-based support. Higher-level interaction models cannot be captured or componentized as is done by *Finesse* modules. EAI toolkits do provide an additional layer above the middleware platform to assist in supporting more complex interactions, but are hampered by limited semantic models for describing interface and interaction behaviour.

The third assertion stated that:

> *Communication networks cannot consistently deliver high-bandwidth, low-*

> *latency, low-failure communications. Distributed software must be able to deal with the degradation or loss of communication.*

The bandwidth, latency and reliability of communication networks are of most concern when synchronous behaviour is required by a set of cooperating application components. In a synchronous interaction, a component cannot continue until an appropriate reply is received from one or more remote components. Unless the network quality of service can be guaranteed, this causes significant performance, reliability and usability degradation. While it is not possible to avoid all synchronization in distributed applications, most distributed software platforms impose additional synchronization requirements, either through forcing components to use a synchronous interaction model or by requiring synchronization to maintain internal state. *Finesse* shows that it *is* possible to avoid imposing synchronization requirements in the distributed software platform. We believe it is unique in doing so amongst implemented distributed software platforms.

In addition to providing a platform with no runtime synchronization requirements, the *Finesse* approach of declaratively specifying event dependencies encourages a programming ethic that avoids unnecessary application-level synchronization and provides significant opportunities for optimising the size and frequency of messaging between distributed components.

9.2 A Critical Examination

The majority of the discussion within and around the previous chapters has focused on the key strengths of *Finesse*. While we believe *Finesse* to be both refined and extremely useful, there are a number of deficiencies. The following subsections describe those deficiencies and where possible, suggest future work that could address those deficiencies.

9.2.1 Complexity

The *Finesse* approach addresses the intrinsically difficult problem of distributed programming. The key difficulty in *Finesse* is in describing parallel behaviours and their synchronization. While the declarative, event-relationship approach is relatively easy

to use where relationships are simple, complex relationships can introduce synchronization requirements that are quite difficult to understand and debug. Traditional semantic models like those used in CSP[59], Lotos[18] and other parallel specification languages rely on an underlying sequential model to simplify reasoning and reduce complexity. Middleware platforms like CORBA[98] rely on static and often synchronous interaction models that hide much of the complexity associated with distributed interaction. We have demonstrated that the *Finesse* approach is more flexible and powerful, but these come at the expense of complexity.

The argument against a more complex model is that programmers should be shielded from the complexity as much as possible. Consider the example of our CSCW classroom program: modules are used to hide the intrinsic complexity associated with the interactions, but someone has to write, debug and understand those modules. We counter that it is possible to provide that shield through use of a modular and/or object-oriented language that allows a programmer to abstract over the complexity in most cases, but also allows an experienced programmer to access the power and flexibility of the underlying model when necessary. This argument is consistent with the assertions of a number of researchers in language and programming technology, in particular, the idea of *Open Implementation* introduced by Kiczales[70] and further promoted by Dourish in his PhD thesis[36]. Open Implementation suggests that complexity can be hidden by language mechanisms, but the underlying complexity should be accessible and modifyable in a semantically consistent manner. Doug Lea[78] also comes out strongly in favour of this approach and backs his assertions with experience in building distributed, object-oriented software. In *Finesse*, the complexity of distributed interaction can be hidden through its modules, and as shown in the examples of chapter 8, those modules can be extended or modified to suit changing requirements.

9.2.2 Finesse Language Syntax

The *Finesse* language syntax has been a vehicle for demonstrating the power and flexibility of the *Finesse* approach. It remains, however, a prototype language and as such has a number of deficiencies. In particular, the *Finesse* language syntax is only partially able to mask the complexity of distributed programming through its notion of

modules. We recognize this deficiency and suggest that future work on the *Finesse* approach should include both visual and object-oriented programming languages based on the same underlying semantic model.

The use of guards rather than more traditional control constructs like `if-then-else` is a questionable aspect of the language syntax. Guards can be more succinct and have a more direct relationship with the semantic model but are often cumbersome, especially where a guard is expressing the negation of a guard on an alternative event path. Future versions of the syntax or alternative syntax might include explicit `if-then-else` constructs, although guards could be retained for generality.

9.2.3 Quality of Service

A significant issue in the *Finesse* approach is highlighted by the streaming data example from chapter 8: specifying quality of service constraints through the explicit introduction of sequence numbers and dependent guards over individual events is cumbersome and difficult to understand. The need to introduce explicit "aggregation" events to keep track of lost packets, for example, is undesirable. While these issues can be addressed to some extent by syntactic constructs, there are more fundamental issues. For example, it is quite difficult to infer quality of service constraints deterministically from the guards on an event-based specification, thus making it difficult to establish communications of quality sufficient to meet the constraints at program instantiation time. A higher-level semantic model for describing quality of service constraints is necessary to support these features.

A related problem is that of "lip-sync". Consider the stream example of chapter 8: we do not specify the synchronization of the video and audio streams. For multimedia applications, this synchronization is very important. While it is possible to achieve this through guards over sequence numbers associated with events, it is again difficult to infer the quality of service requirement from the event-level specification.

The issue of supporting quality of service constraints requires further research and should be directed by the significant body of work in this area. A possible approach would be to introduce language-level constructs that generate both a deterministic quality requirement specification for use at program instantiation time and an event-

level specification that enforces the quality requirements at run-time.

9.2.4 Dynamic Behaviour Instantiation

The *Finesse* semantic model and language have no facility for the dynamic creation of behaviour instances. An implication of this restriction is that parallel iteration is not possible within the current model. The key difficulty is that the execution model requires a static description of the possible behaviour of each participant and how this is related to the behaviour of other participants. If multiple instances of a particular behaviour are created at one participant, the relationship between those instances and equivalent instances of behaviour at other participants needs to be established in a non-deterministic manner. At this point, we have not specified a deterministic mechanism for linking such parallel instances of behaviour across participants.

Given that *Finesse* is already a significant improvement in flexibility and power over existing platforms, we do not see this as a fundamental deficiency. In many senses, the restriction is appropriate in the semantic model because a single participant implies a sequential order over the events occurring at that participant. The problem can be circumvented to some extent by the ability to instantiate a program with any number of participants filling a given role: multiple parallel instances of behaviour become multiple participants in the program. In this case, introducing a new participant implies a re-binding, and although not reported in this thesis, some initial specification and prototyping of dynamic re-binding capability in *Finesse* has begun.

The underlying semantic problem is not insoluble, but any solution introduces an additional level of complexity into the semantics. The semantic problem we describe has been addressed to some extent in the π calculus[94] through the ability to transfer channels, and this will possibly influence future work on *Finesse* in this area. It is also likely that the problem can be addressed through deterministic renaming of events in parallel behaviour instances.

9.2.5 Security

Although not strictly a deficiency in the *Finesse* approach, our claim that *Finesse* is useful for B2B interaction is only accurate if the implementation provides security

mechanisms sufficient for such interaction. There are two key aspects of security that must be addressed, and a third that can usefully be addressed:

1. Participants in a program must be identified (authentication)
2. The actions of participants must be suitably controlled (authorization)
3. It should be possible to determine if an action of a participant was or was not performed (non repudiation)

Interestingly, *Finesse* has characteristics that are useful in the establishment of these mechanisms. The step of instantiating a program requires an unambiguous identification of the parties involved, thus secure authentication can be achieved by securing this process with an appropriate mechanism. The specification of the program to be executed by all parties at instantiation time and the acceptance of the participants fulfilling roles defines the permissible actions of participants, thereby defining the necessary authorization. This mapping between the steps of program instantiation and the necessary security controls makes the implementation of these controls relatively straightforward. In particular, the fact that each copy of the runtime engine operates autonomously and under the control of the participant allows a participant to detect incorrect behaviour that might violate security controls. For interactions with no single controlling authority like B2B, this is particularly important. Note, however, that it is also necessary to ensure the integrity of the program instantiation process, or in other words, ensuring that the process is not being circumvented in any way. This is an issue for future research.

The implementation of non-repudiation is not yet a standard component in many distributed software platforms. A typical implementation involves notifying a trusted third party when any significant actions occur. Since all externally visible actions are explicitly defined in a *Finesse* program, it would be straightforward to parse a program and generate an augmented program that specifies reliable notification to a non-repudiation role (a notary) for a set of nominated events. Instantiation of the augmented program would then include agreement on the identity of the notary and the runtime would automatically implement the necessary notifications.

We conclude that although security is not implemented or specified in this thesis, the program instantiation and execution model lends itself naturally to the implementation of security controls.

9.3 Related Work

Finesse does not solve all distributed systems problems, but it provides a consistent and implementable framework within which solutions to distributed systems problems can be described and implemented in a flexible and re-usable fashion. This is perhaps the most compelling argument in favour of *Finesse*: there is no other system that we are aware of that provides such a powerful framework. That said, *Finesse* has many similarities with existing work both in its detail and its high-level features. The following subsections compare aspects of *Finesse* with relevant existing work. We focus initially on theoretical models and approaches that differ from *Finesse*, then discuss the more concrete programming platforms.

9.3.1 Models for Parallel and Distributed Systems

The comparison of *Finesse* with existing semantic models for parallel and distributed systems has been addressed to some extent in chapter 4. The key point to be taken from that chapter is that while many models provide similar expressive power to *Finesse*, few are so directly implementable by a set of distributed, asyncronous, and autonomous runtime engines.

The nearest theoretical model is that of event structures[134]. The two models are quite similar in their description of an execution of a parallel program. Event structures, however, have remained largely in the realm of mathematical theory and semantic analysis and have rarely been used in the specification of implementable programs. Event structures, however, have been used as an underlying semantic model for Petri nets[97]. Petri nets[99] provide a truly concurrent programming model based on the notions of nodes, tokens, and places. As with *Finesse*, it is possible to distribute the behaviour described by a Petri net over a set of distributed participants. The distribution, however, is subject to synchronization constraints intrinsic to the model. Also,

while Petri nets have found favour in protocol specification, analysis and simulation, it is difficult to map the Petri net semantics onto common interaction mechanisms associated with application components. This has meant, for example, that there are few examples of coordination languages using the Petri net semantic model.

A number of other models for parallel and distributed systems based on interleaved concurrency and sequential trace semantics are popular amongst researchers. This includes CSP[59], CCS[93], Lotos[18], and the π calculus[94]. Their expressiveness in describing parallel applications is similar to *Finesse* . The π calculus in particular is able to describe the transfer of "channels" and thus allows the dynamic instantiation of parallel behaviour, a deficiency in *Finesse*. The key advantage of the *Finesse* semantic model is its native ability to describe true parallism and hence the conflicts that can occur in a distributed system. For example, it is difficult to capture the potential conflict that exists when mutually exclusive parallel events are executable in an interleaved concurrency model: one event can simply be chosen making the other impossible. In a distributed environment, this choice requires synchronization because the events can truly occur in parallel and the *Finesse* model makes this quite clear. Over and above this theoretical advantage, the mapping of the *Finesse* semantic model directly onto a set of distributed, asynchronous runtime engines is a significant result and one that is made easier and clearer by the nature of the model.

9.3.2 Mobile Agent Technology

While one could consider mobile agent technology[28] to be a model for parallel and distributed systems, we address it separately because it is perhaps closest to providing an equivalent distributed and implementable model to *Finesse*. Mobile agent technology is generally characterized by the use of a mobile "agent" that includes code, program state and program data. The agent migrates between participants in a distributed program as defined by the program and the current state of the agent, performing actions against each participant when co-located and updating its program state and data to reflect those actions. By definition, a single agent can only execute a single threaded process or one with only local parallelism. Most agent languages and systems, however, offer the ability to split or create child or clone agents along some synchronization

primitives to coordinate their actions.

The expressive power of mobile agent technology is similar to that provided by *Finesse*. The issue of which model better handles the complexity of truly distributed software is open to argument. The key advantages of *Finesse* over mobile agent technology can be summarized as follows:

- *Finesse* uses a declarative programming model, allowing significant opportunities for optimization. By comparison, the imperative model of agent technology leaves little scope for optimization. The need to transfer the whole program state and data with each migration of the agent can also be a particularly inefficient consumer of bandwidth;

- we are not aware of any models for agent technology that explicitly distinguish roles and allow for multiple participants to fulfill a role. This makes it considerably more difficult to implement, for example, process group abstractions and other mechanisms for reliable distributed computing;

- the explicit maintenance of a view of program state for each participant in *Finesse* increases the autonomy of participants and is particularly useful for monitoring and securing B2B interactions;

- the ability to access local program state at any time in *Finesse* also makes debugging programs considerably easier;

- the model of executing methods against the mobile program state is good for object-oriented programs, but does not match well with streaming data or other interaction models requiring bulk data transfer.

- the transfer of the complete program, state, and data to each participant poses some unique security risks for agent technology[28, 65]. In particular, it is impossible to prevent and difficult to detect participant modifications to the agent state that violate authorization constraints.

The key advantage of mobile agent technology over *Finesse* is that the base behavioural model builds on existing object-oriented programming models, that is, execution of methods on an encapsulated object (the agent). This is well understood by

most programmers and makes agent technology accessible from within existing programming frameworks. We conclude that while this technology provides a useful and more approachable technique for distributed programming, the limitations described above make the *Finesse* approach more flexible and powerful in many circumstances.

Note that these comments are not directed at the agent communication languages associated with intelligent agents[88, 89]. It is also worth noting that agent communication languages, and in particular the KQML language[88], could potentially use the semantic model described in chapter 4 as a basis for execution and the definition of new *performatives* for interaction between agents.

9.3.3 Coordination and Architecture Description Languages

The *Finesse* language is most closely related to a number of existing coordination languages and architecture description languages. As discussed in chapter 3, the coordination community tends to be divided between tuple space models and connection-oriented models. The tuple space models are derived from the Linda[23] prototype, and provide coordination through a set of well defined operations on a logically shared data space. While they provide a basis for interconnection of components, the presentation of that functionality is quite different from *Finesse*. The connection-oriented models are usually based on a language for describing the interconnection of components and are quite similar to *Finesse* in the model and features provided. Quite closely related again are the architecture description languages. These languages are intended to model the high-level architecture of component-based systems, so provide semantic constructs for describing the interconnection of components.

The earliest attempts at distinguishing coordination from computation were based on Linda[23] and a number of variants are still in active use, indicating the power of the shared tuple space approach. These systems have the advantage of a simple, yet powerful model of communication. *Finesse* lacks this simplicity but has a number of advantages, in particular the ability to abstract over communication in a way that can be optimized, and the ability to *capture* coordination protocols and build increasingly high-level abstractions of that coordination. The shared tuple space models suffer in distributed systems because of the implicit synchronization required to maintain a

logically consistent tuple space.

More recently, a number of coordination languages have been based on the idea of building a network of connections between ports and/or interfaces, as is done by ConCoord[60] and Manifold[4]. ConCoord in particular has powerful abstraction capabilities and language independence. The primary difference between these languages and *Finesse* is that *Finesse* does not use explicit connections between interfaces, with the abstracted causal and parameter relationships allowing the optimization of messaging and message contents. Neither the connection based languages or the shared tuple-space languages support real-time constraints to the extent supported by *Finesse*.

Architecture description languages are typically intended to describe the high level structure of a software system for modelling and analysis purposes. They provide similar abstractions and expressive power to the *Finesse* language syntax, but do not offer an underlying distributed execution model or a direct mapping to such a model. Some key examples include Darwin[84] which uses a semantic model based on π calculus, Wright[101] which is based on CSP, and Rapide[80]. Semantically, *Finesse* is most similar to Rapide, whose semantics is based on *posets* (partially ordered sets of events). Rapide is intended as a simulation language for software engineering. It is event-based, with a true concurrency model based on causality, and uses event patterns for abstraction and synchronization. Rapide also has extensive support for real-time constraints. *Finesse* differs most from Rapide in the way abstraction is handled and in its data model, since Rapide has a fixed, structured data model. Work derived from Darwin[74] is now used to generate implementation code making it similar in high-level functionality, but without the distribution semantics.

9.3.4 Middleware Platforms

Section 3.6 of chapter 3 provides an overview of both research and commercial middleware platforms intended to provide a basis for constructing distributed systems. As suggested in that discussion, middleware platforms suffer from the static nature of their interaction mechanisms: it is not generally possible to define new, higher-level interaction protocols, and the systems focus on building tightly-coupled software where

all components are known at compile-time. This static property has led to the recent popularity of EAI systems.

9.3.5 EAI Platforms

EAI platforms have grown from a need to integrate enterprise-level components in large commercial organizations. While they are based on relatively sound distributed systems principles, the primary force behind their development was large consulting organizations faced with the difficulty of building such integrations. There are few, if any, research prototypes offering the features of an EAI platform. Some key competitors in the EAI marketplace include Vitria with BusinessWare, Tibco with ActiveEnterprise, IBM with MQSeries Integrator, and BEA with their Weblogic Integrator. The EAI platforms have remarkably similar architectures:

- application components publish events or requests
- a broker accepts each event or request
- a program or programs in the broker performs one or more actions as a result: this could be as simple as forwarding the event to another application component, or could involve the generation of a set of new events based on a set of event triggers and forwarding these to other application components
- application components also perform actions as a result of receiving events or requests

The broker programs are often graphically defined or scripted and can include data transformation operations. Most platforms can also use a workflow engine to implement business processes that require monitoring and human interaction. Messaging in these platforms is typically available in both reliable and unreliable variants. Many of the platforms are based on a transactional messaging infrastructure.

These toolkits offer many features and tools that are not matched by *Finesse*, but the centralized, broker-based architecture is both a potential performance bottleneck and a single point of failure. There is considerable potential for using *Finesse* as a distributed broker in these platforms. The interface descriptions and broker program

semantics are also quite limited in many cases. For example, Vitria's BusinessWare product supports only publish/subscribe interaction. These systems could benefit significantly from the well-structured, well-defined and extensible approach used in *Finesse* . We conclude by stating that EAI platforms use a sound architecture and provide good tools, but suffer from ad-hoc or inflexible semantic models for interaction.

9.3.6 B2B Platforms

There are a number of existing systems designed to support the development of B2B systems. These include initiatives like RosettaNet, Hewlett Packard's E-speak infrastructure, and Microsoft's BizTalk server. These systems, however, focus on the definition of business documents and the point-to-point transfer of those documents. There is little or no support for the higher-level definition of the roles and responsibilities captured in the business contract. These are primarily left to local programming.

The idea of implementing business contracts in a distributed system has been explored by a number of researchers, including [95, 54]. In both cases, the work focuses on the higher-level issues of establishing contracts and they assume the presence of an underlying execution engine for processes used for contract enactment. The CrossFlow work in particular uses existing workflow technology and is dependent on a centralized engine or synchronized distributed engines. The *Finesse* approach avoids the need for a centralized engine or synchronized distributed engines.

9.4 Concluding Remarks

This chapter has presented a critical examination of *Finesse* with the goal of highlighting its strengths and weaknesses, and showing how *Finesse* compares with the capabilities of existing systems. From this discussion, we assert that *Finesse* does satisfy the needs of distributed systems defined by the initial assertions of the thesis, and provides a number of unique advantages in the construction of distributed systems.

Chapter 10

Conclusion

This thesis has presented *Finesse* as a new approach to the construction of distributed systems. The approach draws together both theory and practice from software architecture, distributed systems, parallel software specification languages, coordination languages, and middleware technology. While individual pieces of the *Finesse* are significant in themselves, the major contribution is the totality of the approach.

We began with an architectural model for the interconnection of distributed components. This model separates interface from interaction and promotes the interaction specification to first-class status. Or in other words, we support the notion of truly programmable middleware. We then added a behavioural model based on declarative, causal relationships between events, and defined an executable semantic model that allows the execution of a program to be distributed amongst a set of autonomous participants. The distributed execution of the program proceeds in an entirely asynchronous manner, with no synchronization except that explicitly specified in the program. We defined a language syntax that links the architectural and behavioural models by specifying the externally visible behaviour of application components (their interfaces) using events, then linking those behaviours through an interactions specification. We also described the design of a prototype implementation of the runtime system, thus showing the implementability of *Finesse* . We then demonstrated the power and flexibility of *Finesse* through a set of examples spanning a variety of domains in distributed systems programming.

In the discussion of chapter 9, we showed that *Finesse* capably meets the goals

for distributed systems programming set down in the introductory comments. Our critique of the approach identifies shortcomings in *Finesse* and suggests how they can be addressed by future work. We also presented a comparison between *Finesse* and existing work, demonstrating in the process that the independent pieces of *Finesse* are useful innovations in themselves. In isolation, the key innovations of the *Finesse* approach can be summarized as follows:

- the specification of behaviour using a declarative model based on causal relationships;
- the definition of an executable semantic model for that behaviour that allows a program to be distributed amongst a set autonomous participants communicating only through asynchonous messaging;
- the definition of a simple but flexible and powerful programming language syntax for creating programs using the declarative model;
- the presentation of a design and prototype implementation of a distributed runtime engine that can execute programs using the model.

We emphasize, however, that although these innovations are significant, the totality of the approach is the key contribution of the thesis.

This thesis is submitted with confidence that the *Finesse* system is a novel, pragmatic and powerful approach to the construction of distributed systems, and will provide a basis for both ongoing research and commercial implementation.

Bibliography

[1] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings 16th International Conference on Software Engineering*. IEEE, May 1994.

[2] J. Andrade, M. Carges, T. Dwyer, and S. Felts. *The Tuxedo System: Software for Constructing and Managing Distributed Business Applications.* Addison Wesley, 1996.

[3] J. Andreoli, S. Freeman, and R. Pareschi. The coordination language facility. *Theory and Practice Of Object Systems*, 2(2):77–94, 1996.

[4] F. Arbab. The IWIM model for coordination of concurrent activities. In *Coordination Languages and Models*, number 1061 in LNCS. Springer, 1996.

[5] F. Arbab. The influence of coordination on program structure. In *Proceedings of the Thirtieth Annual Hawaii International Conference on System Sciences: Software Technology and Architecture*, 1997.

[6] D. Arnold, A. Bond, M. Chilvers, and R. Taylor. Hector: Distributed objects in python. In *Proceedings of the 4th International Python Conference*, Livermore, California, June 1996.

[7] H. Bal, M. Kaashoek, and A. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3), Mar. 1992.

[8] B. Berliner. CVS II: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*. USENIX Association, 1990.

[9] A. Berry and S. Kaplan. Open, distributed coordination with finesse. In *ACM Symposium on Applied Computing*, Atlanta, Feb. 1998.

[10] A. Berry and S. Kaplan. A distributed asynchronous execution semantics for programming the middleware machine. In *Fifth International Symposium on Autonomous Decentralized Systems*, Dallas, Texas, USA, Mar. 2001. IEEE.

[11] A. Berry and K. Raymond. The A1$\surd$ architecture model. In *Open Distributed Processing: Experiences with distributed environments*. IFIP, Chapman and Hall, Feb. 1995.

[12] K. Birman. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. *Operating Systems Review*, 28(1):11–20, Jan. 1994.

[13] K. Birman and R. Cooper. The ISIS project: Real experience with a fault tolerant programming system. *Operating Systems Review*, Apr. 1991.

[14] K. P. Birman. Maintaining consistency in distributed systems. Technical report, Cornell University, Nov. 1991.

[15] A. Birrel and D. Nelson. Implementing remote procedure calls. *ACM Transactions on Computing Systems*, 2(1), Feb. 1984.

[16] A. Black, N. Hutchison, E. Jul, and H. Levy. Object structure in the Emerald system. In *Proceedings of Object Oriented Programming Systems, Languages and Applications*, volume 21 of *SIGPLAN Notices*, Nov. 1986.

[17] G. Blair and T. Rodden. The challenges of CSCW for Open Distributed Processing. In *Open Distributed Processing, II*. IFIP, North Holland, 1993.

[18] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[19] G. Booch. *Object Oriented Design With Applications.* Benjamin/Cummings, Redwood City, California, 1991.

[20] W. Brookes, J. Indulska, A. Berry, K. Raymond, and B. A. A type model supporting interoperability in open distributed systems. In *Proceedings of the*

Telecommunications Information Networking Architecture Conference, Melbourne, Feb. 1995.

[21] N. Carriero, D. Gelernter, and S. Hupfer. Collaborative applications experience with the Bauhaus coordination language. In *Proceedings of the Thirtieth Annual Hawaii International Conference on System Sciences: Software Technology and Architecture*, 1997.

[22] N. Carriero, D. Gelernter, and L. Zuck. Bahaus linda. In *Object-based Models and Languages for Concurrent Systems*, number 924 in LNCS, pages 66–76. Springer, July 1994.

[23] N. Carriero and G. Gelernter. Linda in context. *Communications of the ACM*, 32(4), Apr. 1989.

[24] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems*. McGraw Hill, 1984.

[25] B. Charron-Bost, C. Delporte-Gallet, and H. Fauconnier. Local and temporal predicates in distributed systems. *ACM Transactions on Programming Languages and Systems*, 17(1), 1995.

[26] D. Cheriton. The V kernel: A software base for distributed systems. *IEEE Software*, Apr. 1984.

[27] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communications. *Operating Systems Review*, 27(5):44–57, December 1993. Proceedings of Fourteenth ACM Symposium on Operating Systems Principles.

[28] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik. Itinerant Agents for Mobile Computing. *IEEE Personal Communications*, 2(5):34–49, 1995.

[29] D. Chess, C. Harrison, and A. Kershenbaum. Mobile Agents: Are They a Good Idea? Technical Report RC 19887, IBM, Yorktown Heights, New York, 1994.

[30] P. K. Chrysanthis and K. Ramamritham. Acta: A framework for specifying and reasoning about transaction structure and behavior. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 194–203. ACM, May 1990.

[31] P. Ciancarini and C. Hankin, editors. *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*. Springer, 1996.

[32] K. Clark and S. Gregory. PARLOG: Parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1), Jan. 1986.

[33] M. Cortes and P. Mishra. DWCPL: A programming language for describing collaboration. In *ACM 1996 Conference on Computer Supported Cooperative Work*, Nov. 1996.

[34] E. Denti, A. Natali, and A. Omicini. On the expressive power of a language for programming coordination. In *Symposium on Applied Computing*. ACM, 1998.

[35] P. Dourish. A divergence-based model of synchrony and distribution in collaborative systems. Technical report, Rank Xerox Research Centre, Cambridge Laboratory, 1994.

[36] P. Dourish. *Open Implementation and Flexibility in CSCW Toolkits*. PhD thesis, Department of Computer Science, University College London, 1996.

[37] H. Edelstein. Unraveling client-server architectures. *DBMS*, 7(5), may 1994.

[38] K. Edwards. Session management for collaborative applications. In *Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work*, Oct. 1994.

[39] M. Fazzolare, B. G. Humm, and R. D. Ranson. Advanced transaction semantics for TINA. In *Proceedings of the Fourth Telecommunication Information Networking Architecture Workshop*, pages 47–57, Sept. 1994.

[40] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, pages 28–33, Aug. 1991.

[41] G. Fitzpatrick, S. Kaplan, and T. Mansfield. Physical spaces, virtual places and social worlds: A study of work in the virtual. In *Proceedings of ACM 1996 Conference on Computer Supported Cooperative Work*, pages 334–343, Boston, MA, 1996. ACM Press.

[42] G. Fitzpatrick, W. J. Tolone, and S. M. Kaplan. Work, locales and distributed social wOrlds. In H. Marmolin, Y. Sundblad, and K. Schmidt, editors, *Proceedings of the Fourth European Conference on Computer-Supported Cooperative Work*, pages 1–16. Kluwer Academic Publishers, 1995.

[43] R. Frederick. Experiences with software real time video compression. Technical report, Xerox PARC, 1992.

[44] R. Furuta and P. D. Stotts. Interpreted collaboration protocols and their use in groupware prototyping. In *Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work*, Oct. 1994.

[45] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design issues. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering*. ACM Press, Dec. 1994.

[46] D. Gelernter. Generative communications in linda. *ACM Transactions on Programming Languages and Systems*, pages 80–112, Jan. 1985.

[47] D. Georgabopoulos, M. Hornick, and P. Krychniak. An environment for specification and managment of extended transactions in DOMS. In *Proceedings of the 3rd International Workshop on Interoperable Multidatabase Systems*, Apr. 1993.

[48] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modelling to workflow automation infrastructure. *Distributed and Parallel Databases*, 1995.

[49] D. Gifford. Weighted voting for replicated data. In *Proceedings* 7th *Symposium on Operating System Principles*, pages 150–161, Pacific Grove, 1979. ACM.

[50] L. Gilman and R. Schreiber. *Distributed computing with IBM MQSeries*. Wiley, 1996.

[51] J. Gray. *Operating Systems: An Advanced Course*. Lecture Notes in Computer Science. Springer-Verlag, 1978.

[52] S. Greenberg and D. Marwood. Real time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work*. ACM Press, Oct. 1994.

[53] C. Greenhalgh and S. Benford. MASSIVE: a distributed virtual reality system incorporating spatial trading. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, May 1995.

[54] P. Grefen, K. Aberer, Y. Hoffner, and H. Ludwig. CrossFlow: Cross-organizational workflow management in dynamic virtual enterprises. *International Journal of Computer Systems Science and Engineering*, 15(5), 2000.

[55] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.

[56] S. Harrison and P. Dourish. Re-place-ing space: The roles of place and space in collaborative systems. In M. S. Ackerman, editor, *Proceedings of ACM 1996 Conference on Computer Supported Cooperative Work*, pages 67–76, Boston MA, Nov 1996. ACM Press.

[57] J. Heidemann, T. Page, R. Guy, and G. Popek. Primarily disconnected operation: Experience with Ficus. In *The 2nd International Workshop on Mangement of Replicated Data*, nov 1992.

[58] R. Helm, I. Holland, and D. Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. *SIGPLAN Notices*, 25(10):169–180, 1990.

[59] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[60] A. A. Holzbacher. A software environment for concurrent coordinated programming. In *Coordination Languages and Models*, volume 1061 of *LNCS*. Springer, 1996.

[61] J. Howard. An overview of the Andrew File System. In *USENIX Winter Technical Conference*, 1988.

[62] N. Islam and R. H. Campbell. Techniques for global optimization of message passing communication on unreliable networks. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, May 1995.

[63] ISO/IEC 10746-1 10756-2 10746-3 10746-4 Basic Reference Model for Open Distributed Processing.

[64] V. Jacobson and S. McCanne. Vat—X11-based audio teleconferencing tool. Unix man pages, Lawrence Berkeley Laboratory, 1993.

[65] W. Jansen. Countermeasures for mobile agent security. Technical report, National Institute of Standards and Technology, Gaithersburg, MD, USA, 1999.

[66] B. Janssen and M. Spreitzer. ILU: Inter-language unification via object modules. In *OOPSLA '94 Workshop on Multi-Language Object Models*, 1994.

[67] S. Kaplan, G. Fitzpatrick, T. Mansfield, and W. J. Tolone. MUDdling through. In *Proceedings of the Thirtieth Annual Hawaii International Conference on System Sciences: Information Systems—Collaboration Systems and Technology*, 1997.

[68] S. M. Kaplan, G. Fitzpatrick, and T. Mansfield. Orbit and support for pervasive collaboration. In J. Grundy, editor, *Proceedings of the OzCHI'96 Workshop on the Next Generation of CSCW Systems*, pages 10–14. Dept. of Computer Science, The University of Waikato, New Zealand, 1996.

[69] S. M. Kaplan, W. J. Tolone, D. P. Bogia, and C. Bignoli. Flexible, active support for collaborative work with ConversationBuilder. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 378–385, 1992.

[70] G. Kiczales. Beyond the black box: Open implementation. *IEEE Software*, pages 8–11, Jan. 1996.

[71] T. Kielmann. Designing a coordination model for open systems. In *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*. Springer, 1996.

[72] K. Klockner, P. Mambrey, M. Sohlenkamp, W. Prinz, and L. Fuchs. POLITeam: Bridging the gap between bonn and berlin for and with the users. In *Proceedings of the Fourth European Conference on Computer-Supported Cooperative Work*, pages 17–31. Kluwer, Sept. 1995.

[73] Q. Konq and A. Berry. A general resource discovery system for Open Distributed Processing. In *Open Distributed Processing: Experiences with distributed environments*. IFIP, Chapman and Hall, Feb. 1995.

[74] J. Kramer and J. Magee. Exposing the skeleton in the coordination closet. In *Coordination Languages and Models*, number 1282 in Lecture Notes in Computer Science, pages 18–31. Springer, Sept. 1997.

[75] M. I. Kramer. Business events: Publish and subscribe technology enables automation through application integration. *Distributed Computing Monitor*, 10(11):3–27, Nov. 1995.

[76] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawhat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, Nov. 92.

[77] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, pages 558–565, July 1978.

[78] D. Lea. Design for open systems in java. In *Coordination Languages and Models*, number 1282 in Lecture notes in computer science. Springer, 1997.

[79] B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3), Mar. 1988.

[80] D. C. Luckham and J. Vera. An event based architecture definition language. *IEEE Transactions on Software Engineering*, Sept. 1995.

[81] M. Lutz. *Programming Python*. O'Reilly and Associates, 1996.

[82] S. Maffeis. The electra approach to object oriented distributed programming. Technical Report IFI TR 92.23, Institut fur Informatik Der Universitat Zurich, Nov. 1992.

[83] S. Maffeis. iBus: The java intranet software bus. Technical report, SoftWired AG, Zurich, Switzerland, 1997.

[84] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, Sept., 1995.

[85] D. Marshak. Lotus Notes: A platform for developing workgroup applications. *Patricia Seybold's Office Computing Report*, July 1990.

[86] D. S. Marshak. ANSA: A model for distributed computing. *Network Monitor*, 6(11), Nov. 1991.

[87] A. Mathur, R. Hall, J. Farnam, A. Prakesh, and C. Rasmussen. The publish/subscribe paradigm for scalable group collaboration systems. CSE-TR 270-95, University of Michigan, Nov. 1995.

[88] J. Mayfield, Y. Labrou, and T. Finin. Evaluation of KQML as an Agent Communication Language. In *Proceedings on the IJCAI Workshop on Intelligent Agents II : Agent Theories, Architectures, and Languages*, volume 1037, pages 347–360. Springer-Verlag, 1996.

[89] F. G. McCabe and K. L. Clark. April – agent process interaction language. In *Intelligent Agents: Theories, Architectures, and Languages (LNAI volume 890)*, pages 324–340. Springer-Verlag, 1995.

[90] S. McCanne and V. Jacobson. Vic: A flexible framework for packet video. In *ACM Multimedia*, nov 1995.

[91] R. Medina-Mora, T. Winograd, R. Flores, and F. Flores. The action workflow approach to workflow management technology. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 281–288, 1992.

[92] Microsoft Corporation. *Microsoft Networks SMB File Sharing Protocol*. Also known as CIFS.

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

[94] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Journal of Information and Computation*, 100, 1992.

[95] Z. Milosevic, A. Berry, A. Bond, and K. Raymond. Supporting business contracts in open distributed systems. In *Proceedings of the Workshop on Services in Distributed and Networked Environments*. IEEE, 1995.

[96] NCSA. Habanero. http://www.ncsa.uiuc.edu/SDG/Software/Habanero/.

[97] M. Neilsen, G. Plotkin, and W. G. Petri nets, event structures and domains. In *Semantics of Concurrent Computation*, volume 70. Springer-Verlag, 1979.

[98] The common object request broker: Architecture and specification. The Object Management Group, 1995. Revision 2.0.

[99] Peterson. *Petri net theory and the modeling of systems*. Prentice-Hall, 1981.

[100] L. L. Peterson. Preserving context information in an IPC abstraction. In *Proceedings of the 6th symposium on Reliability in Distributed Software and Database Systems*, pages 22–31, Mar. 1987.

[101] A. R. and D. Garlan. Formal connectors. Technical Report CMU-CS-94-115, Carnegie Mellon University, Mar. 1994.

[102] A. Rakotonirainy, A. Berry, S. Crawley, and Z. Milosevic. Describing open distributed systems: A foundation. *The Computer Journal*, 40(8), 1997.

[103] A. Rakotonirainy, A. Berry, S. Crawley, and Z. Milosevic. Describing open distributed systems: A foundation. In *Proceedings of the Thirtieth Annual Hawaii*

International Conference on System Sciences: Software Technology and Architecture, 1997.

[104] Rational. The unified modelling language. http://www.rational.com/uml/.

[105] K. Raymond. Reference Model of Open Distributed Processing (RM-ODP): Introduction. In *Open Distributed Processing: Experiences with distributed environments*. IFIP, Chapman and Hall, Feb. 1995.

[106] M. Rice and S. B. Seidman. A formal model for module interconnection languages. *IEEE Transactions on Software Engineering*, 20(1):88–101, Jan. 1994.

[107] D. Rogerson. *Inside DCOM*. Microsoft Press, 1997.

[108] M. Roseman and S. Greenberg. GROUPKIT a Groupware Toolkit for Building Real-Time Conferencing Applications. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 43–50, 1992.

[109] M. Roseman and S. Greenberg. TeamRooms: Network places for collaboration. In *ACM 1996 Conference on Computer Supported Cooperative Work*, Nov. 1996.

[110] W. Rosenbury, D. Kenney, and G. Fisher. *Understanding DCE*. O'Reilly and Associates, Inc., September 1992.

[111] A. Rowstron and A. Wood. Bonita: A set of tuple space primitives for distributed coordination. In *Proceedings of the Thirtieth Annual Hawaii International Conference on System Sciences: Software Technology and Architecture*, 1997.

[112] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[113] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementataion of the Sun network filesystem. In *Proceedings of the summer 1985 USENIX Conference*, jun 1985.

[114] M. Satyanarayanan, J. Kistlyer, P. Kumar, M. Okasiki, E. Siegel, and D. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4), Apr. 1990.

[115] C. Schuckmann, L. Kirchner, J. Schummer, and J. M. Haake. Designing object-oriented synchronous groupware with COAST. In *ACM 1996 Conference on Computer Supported Cooperative Work*, Nov. 1996.

[116] M. Shaw. Patterns for software architectures. In *Proceedings of First Annual Conference on the Pattern Languages of Programming*, Aug. 1994.

[117] M. Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. Technical Report CMU-CS-94-107, Software Engineering Institute, Carnegie Mellon Univerity, Jan. 1994.

[118] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an emerging discipline.* Prentice Hall, 1996.

[119] M. Sherman. Architecture of the encina distributed transaction processing family. In *ACM SIGMOD Conference*, 1993.

[120] M. Sohlenkamp and G. Chwelos. Integrating communication, cooperation, and awareness: the DIVA virtual office environment. In *Proceedings ACM Conference on Computer Supported Cooperative Work*, pages 331–343, Chapel Hill, NC, 1994. ACM Press.

[121] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1989.

[122] A. Strauss. *Continual Permutations of Action.* Aldine De Gruyter, New York, 1993.

[123] Sun Microsystems. *The Javabeans 1.01 Specification.* http://java.sun.com/products/javabeans/docs/spec.html.

[124] Sun Microsystems. *Network Programming Guide*, Mar. 1990. Part number: 800-3850-10.

[125] V. Sunderam. A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4), 1990.

[126] C. Szyperski. *Component Systems–Beyond Object-oriented Programming*. Addison Wesley, 1998.

[127] A. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. Mullender, A. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(2), Dec. 1990.

[128] D. Terry, M. Theimer, K. Peterson, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, dec 1995.

[129] W. J. Tolone, S. M. Kaplan, and G. Fitzpatrick. Specifying dynamic support for collaborative work within wOrlds. In *Proceedings ACM Conference on Organizational Computing Systems (COOCS'95)*, pages 55–65, Milpitas, CA, 1995.

[130] R. van Renesse, K. P. Birman, and S. Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, Apr. 1996.

[131] J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 1999.

[132] M. Wilkes and R. Needham. The Cambridge model distributed system. *Operating Systems Review*, 14(1), 1980.

[133] T. Winograd and F. Flores. *Understanding Computers and Cognition: A New Foundation for Design*. Addison Wesley, Reading, 1986.

[134] G. Winskel. An introduction to event structures. In *Linear time, branching time and partial order in logics and models for concurrency*, number 354 in LNCS. Springer-Verlag, 1988.

[135] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[136] X/Open. *Distributed Transaction Processing: Reference Model.* X/Open Company Ltd, 1991.

Appendix A

Finesse Language Syntax

The following text presents a BNF specification of the syntax of the Finesse language described in chapter 5. The syntax has been generated from a parser definition using the JavaCC suite of parsing tools from Sun. All complete examples presented in this thesis have been successfully parsed by the prototype parser that generated this specification.

```
       BINDING ::= "Binding" <NAME> "{" BINDDEF "}" <EOF>
       BINDDEF ::= IMPORTS ROLES INTERACTIONS
       IMPORTS ::= ( "Import" <NAME> ( "," <NAME> )* ";" )*
         ROLES ::= "Roles" "{" ( ( CARD )? <NAME>
                     ( NAMELIST )?
                     "{" ( ROLEBEHAVIOUR )? "}" )+ "}"
          CARD ::= "[" <CARDVAL> COMPARE <INTEGER>
                    ( ( <AND> | <OR> | <XOR> )
                      <CARDVAL> COMPARE <INTEGER> )? "]"
 ROLEBEHAVIOUR ::= ROLEACTION ( INFIX ROLEACTION )*
    ROLEACTION ::= ( GUARD )?
                     ( EVENTSPEC
                       | <NAME> "{" ROLEBEHAVIOUR "}"
                       | IMPORTEDACTION )
                   | LOOPACTION
                   | "{" ROLEBEHAVIOUR "}"
IMPORTEDACTION ::= <DOTNAME> ( SUBSLIST )?
     EVENTSPEC ::= <EVENT> ( PARSET ( PARMREL )? )?
        PARSET ::= "(" ( PARSPEC ( "," PARSPEC )* )? ")"
       PARSPEC ::= <NAME> ( ":" <NAME> )?
    LOOPACTION ::= ( <WHILE> GUARD "{" ROLEBEHAVIOUR "}" )
                   | ( <LOOP> "{" ROLEBEHAVIOUR "}" )
      NAMELIST ::= "(" <NAME> ( "," <NAME> )* ")"
```

```
       SUBSLIST ::= "(" ( ( VALUE | PARSET )
                          ( "," ( VALUE | PARSET ) )* )? ")"
   INTERACTIONS ::= "Interactions" "{" INTERACTBEHAV "}"
  INTERACTBEHAV ::= ( INTERSPEC ( INFIX INTERSPEC )* )?
      INTERSPEC ::= ( GUARD )? ( INTERSPECACTION | BINDSPEC )
       BINDSPEC ::= <NAME> "(" ( BOUNDNAME | BOUNDNAMELIST )
                    ( "," ( BOUNDNAME | BOUNDNAMELIST ) )* ")"
  BOUNDNAMELIST ::= "(" BOUNDNAME ( "," BOUNDNAME )* ")"
INTERSPECACTION ::= ( <DOTNAME> ( PARMREL )? )
                    | "{" INTERACTBEHAV "}"
        PARMREL ::= "{" ( ( ASSIGN ( "," ASSIGN )* )
                          | NAMEEQUIV ) "}"
         ASSIGN ::= <NAME> "=" VALUE
       PREVSPEC ::= <PREVPARM> | <DOTNAME>
      NAMEEQUIV ::= "*=" ( <PREV> | <DOTNAME> )
          GUARD ::= "[" GUARDEXPR "]"
      GUARDEXPR ::= GUARDVALUE
                    ( ( <AND> | <OR> | <XOR> )
                      GUARDVALUE )*
     GUARDVALUE ::= ( <NOT> )?
                    ( "(" GUARDEXPR ")"
                       | ( VALUE ( COMPARE VALUE )? )
                       | CARDGUARD )
   BUILTINGUARD ::= TIMEGUARD | REPLYGUARD | OCCURGUARD
      TIMEGUARD ::= ( <TIMELESS> | <TIMEMORE> )
                    "(" EVENTREF "," VALUE ")"
     REPLYGUARD ::= <REPLYTO> "(" EVENTREF "," EVENTREF ")"
     OCCURGUARD ::= <OCCUR> "(" EVENTREF ")"
      CARDGUARD ::= "#" COMPARE ( VALUE | <ALLCARD> )
       EVENTREF ::= <NAME> | <DOTNAME> | <PREV>
          VALUE ::= FUNCTION | BUILTINGUARD
                    | <NAME> | <DOTNAME> | <PREVPARM>
                    | <REAL> | <INTEGER> | <BOOLEAN>
        COMPARE ::= <LESS> | <LESSEQUAL> | <EQUAL>
                    | <GREATER> | <GREATEREQUAL>
       FUNCTION ::= <NAME> "(" ( VALUE ( "," VALUE )* )? ")"
          INFIX ::= <AND> | <OR> | <XOR> | <CAUSES>
      BOUNDNAME ::= <NAME> | <DOTNAME>
```