A Concurrent Object Coordination Language: Semantics and Applications

by

Gordon Wayne O’Connell

B.Sc., University of Victoria, 1976
B.Sc., University of Victoria, 1991
M.Sc., University of Victoria, 1997

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

We accept this dissertation as conforming to the required standard

Dr. Mantis Cheng, Supervisor (Department of Computer Science)

Dr. Gholamali C. Shoja, Departmental Member (Department of Computer Science)

Dr. Bruce Kapron, Departmental Member (Department of Computer Science)

Dr. Issa Traoré, Outside Member (Department of Electrical and Computer Engineering)

Dr. William Older, External Examiner (463 Bolton Rd., Merrickville, Ontario K0G 1N0)

© Gordon Wayne O’Connell, 2004

University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.
Supervisor: Dr. M. Cheng

Abstract

The *Timed Actor model* is a concurrent coordination model which extends the basic *Actor model* of [1] by providing timers and facilities for atomic message processing with *local checkpointing*. The semantics of our *Timed Actor model* allows the specification of a *timed actor language* called COOL, and a virtual machine to accommodate our *timed actor language*. COOL is the basis of our approach for modelling and implementing distributed applications.

COOL provides a high-level description language for specifying the interface, behaviour and coordination of actors. COOL allows a practitioner to specify the *expected behaviour* of actors by defining the performance and coordination properties of actors. These *checkable properties* are compilable into *online monitors*. At run-time monitored activity (*observed behaviour*) is logged in a format amenable to *offline trace analysis*. Our trace analyzer validates a COOL specification by comparing observed to expected behaviour.

Examiners:

---

Dr. Mantis Cheng, Supervisor (Department of Computer Science)

Dr. Gholamali C. Shoja, Departmental Member (Department of Computer Science)

Dr. Bruce Kapron, Departmental Member (Department of Computer Science)

Dr. Issa Traoré, Outside Member (Department of Electrical and Computer Engineering)

Dr. William Older, External Examiner (463 Bolton Rd., Merrickville, Ontario K0G 1N0)
Table of Contents

Abstract ii

Table of Contents iii

List of Tables ix

List of Figures xi

List of Listings xiv

Acknowledgments xvi

1 Introduction 1

1.1 An Approach for Implementing Distributed Systems 2

1.1.1 Modelling and Specification 3

1.1.2 Implementation 5

1.1.3 Validation 7

1.2 Example: A Distributed Token 9

1.3 Related Work 15

1.3.1 Coordination Models and Languages 15

1.3.2 The Actor Model 16

1.3.3 Actor Models with Activation Constraints 19

1.3.3.1 Synchronizers 19

1.3.3.2 The Semantics of Real-Time Constraints 20

1.3.3.3 Actor-Based Real-Time Coordination Languages 21
### 1.3.4 COOL Case Studies .................................................. 21
1.4 Contributions ......................................................... 22
1.5 Organization of the Dissertation .................................... 24

#### 2 Actor Systems ......................................................... 26

2.1 Basic Actor Model ........................................... 28
2.1.1 Actor Instances ........................................... 28
2.1.2 Messages .................................................. 28

2.2 Timed Actor Model ............................................ 29
2.2.1 Timer Instances ........................................... 29
2.2.2 Actor Behaviour ........................................... 30
2.2.3 Timed Actor Configurations ........................... 32
2.2.4 Timer Behaviour ........................................... 35

2.3 STAL—A Simple Timed Actor Language ........................ 36

2.4 Example: Designing a Vending Machine Configuration ........... 40

2.5 Summary .......................................................... 46

#### 3 An Asynchronous Actor Algebra .................................. 47

3.1 A Grammar for $ACube$ Expressions ............................. 48
3.1.1 Syntactic Domains ........................................... 49
3.1.2 Actor Expressions ........................................... 50
3.1.3 Actor Class Definitions .................................... 51

3.2 A Transition System for Actor Expressions ........................ 53

3.3 Static Operators ................................................ 54
3.3.1 Composition ................................................ 54
3.3.2 Association ................................................ 55

3.4 Commitment ...................................................... 57

3.5 Asynchronous Communications .................................... 59
3.5.1 Transition Rules Accept and Reject ......................... 59
3.5.2 The Nature of Actor and Timer Communication ............. 59
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.3</td>
<td>Message Transmission Operations</td>
<td>60</td>
</tr>
<tr>
<td>3.6</td>
<td>Actor Creation</td>
<td>61</td>
</tr>
<tr>
<td>3.7</td>
<td>Semantics of Timers</td>
<td>62</td>
</tr>
<tr>
<td>3.8</td>
<td>Auxiliary Operations</td>
<td>64</td>
</tr>
<tr>
<td>3.8.1</td>
<td>Accessing Local State and Message Parameters</td>
<td>64</td>
</tr>
<tr>
<td>3.8.2</td>
<td>The Conditional Operator</td>
<td>65</td>
</tr>
<tr>
<td>3.9</td>
<td>Exception Handling</td>
<td>65</td>
</tr>
<tr>
<td>3.10</td>
<td>Example: An ACube Definition for the Vending Machine Configuration</td>
<td>67</td>
</tr>
<tr>
<td>3.11</td>
<td>Summary</td>
<td>70</td>
</tr>
</tbody>
</table>

4 A Virtual Machine for a Timed Actor Language  

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Models of Execution</td>
<td>77</td>
</tr>
<tr>
<td>4.2</td>
<td>The Actor Template</td>
<td>78</td>
</tr>
<tr>
<td>4.3</td>
<td>A Dispatcher for a VM Implementing Timed Actor Semantics</td>
<td>81</td>
</tr>
<tr>
<td>4.4</td>
<td>The Dispatcher-Actor Interface</td>
<td>84</td>
</tr>
<tr>
<td>4.4.1</td>
<td>vmdispatch—An Operation to Dispatch a Ready Actor</td>
<td>85</td>
</tr>
<tr>
<td>4.4.2</td>
<td>vmprocess—An Operation for Processing an Intentions List</td>
<td>86</td>
</tr>
<tr>
<td>4.4.3</td>
<td>vmnotify—Reporting Exceptions</td>
<td>87</td>
</tr>
<tr>
<td>4.5</td>
<td>The Actor-Kernel Interface</td>
<td>90</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Actor Creation Operation</td>
<td>91</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Timer Management Operations</td>
<td>92</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Message Transmission Operations</td>
<td>94</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Commitment Operations</td>
<td>97</td>
</tr>
<tr>
<td>4.6</td>
<td>Summary</td>
<td>97</td>
</tr>
</tbody>
</table>

5 COOL—A Timed Actor Language  

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>A Syntax and Informal Semantics of COOL Specifications</td>
<td>102</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Specifications and Classes</td>
<td>102</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Method Blocks</td>
<td>104</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Type Declarations</td>
<td>105</td>
</tr>
</tbody>
</table>
# List of Tables

1.1 Operations for expressing actor behaviour in the *Timed Actor model* .......................... 3
1.2 Operations for expressing actor behaviour in COOL .................................................. 6
1.3 Transition relation for $m \in \text{Token}$ ................................................................. 10
1.4 General properties of actor models ................................................................. 18
2.1 Transition relation for timer $t \in \text{Timer}$ ......................................................... 35
2.2 Transition relation for actor $u \in \text{User}$ ............................................................. 43
2.3 Transition relation for actor $v \in \text{Vend}$ ............................................................. 44
2.4 Example activation history for the vending machine configuration ......................... 45
3.1 Object instances stored on the intentions list ......................................................... 57
4.1 States and transitions describing the behaviour of Dispatcher ............................... 82
4.2 A Dispatcher-actor interface ................................................................................ 85
4.3 Exceptions reported by operation vmnotify ........................................................... 88
4.4 Actor-kernel interface .................................................................................... 90
5.1 Protected instance variables ................................................................................... 107
5.2 Predefined COOL operators ............................................................................... 118
5.3 A comparison of *ACube* operators and COOL statements ................................. 120
6.1 Checkable properties that can be expressed as temporal logic formulae ................ 133
6.2 Data sets required to instrument actor class $C$ ..................................................... 145
6.3 Summary of checkable property implementation and analysis .............................. 155
6.4 The effect of monitoring on response times in Application DABP. .................. 157

7.1 Characteristics of hosts used in case studies. ................................. 162
7.2 Components and input sets for application DABP. .............................. 164
7.3 Application DABP—transition relation for actor instance \( s \in \text{Sender} \). .......... 165
7.4 Application DABP—transition relation for actor instance \( r \in \text{Receiver} \). .......... 166
7.5 Analysis results for application DABP. ........................................ 172
7.6 Components and input sets for the DLOCK simulation. ......................... 176
7.7 Application DLOCK, transition relations for actor \( u \in \text{User} \). ................. 178
7.8 Application DLOCK, transition relations for \( d \in \text{DLock} \). ..................... 180
7.9 Application DLOCK, transition relation for a lock coordinator \( l \in \text{Locker} \). ........ 182
7.10 Some characteristics of application DLOCK. ...................................... 188

A.1 Transition relation for actor \( g \in G \). ........................................ 205

B.1 Syntactic domains used to describe the kernel. ............................... 209
B.2 Functions used in the description of the kernel. ............................... 214
B.3 Pseudocode features for describing a kernel. .................................... 215
B.4 Pseudocode specification for a virtual machine supporting timed actor semantics. 216

D.1 COOL specifications for case studies. ........................................... 227
D.2 Enumeration characteristics of application DLOCK. ........................... 238
List of Figures

1.1 Modelling and specification. ............................. 5
1.2 Instrumenting and building an implementation. ............. 7
1.3 Executing and monitoring an implementation. .................. 8
1.4 Communicating finite state machine for $m \in \text{Token}$. .............. 10
1.5 A message sequence chart illustrating competition for a distributed token. .... 11
1.6 Organization of our research. ................................ 23

2.1 The relationship between actor behaviour and changes in a configuration. ...... 34
2.2 Communicating finite state machine for $t \in \text{Timer}$. .................. 36
2.3 A vending machine configuration. ................................ 41
2.4 A message sequence chart for a timed vending machine. .................. 42

3.1 $A\text{Cube}$ actor expression for class $\text{User}$. .......................... 49
3.2 An operational semantics for $A\text{Cube}$ static operators. ................. 56
3.3 An operational semantics for $A\text{Cube}$ dynamic operators. ............... 66
3.4 A vending machine configuration with bootstrap actor $m \in \text{Main}$. ......... 67
3.5 $A\text{Cube}$ definition for a vending machine configuration. .................. 68
3.6 Vending machine configuration—processing a typical input sequence. ....... 69
3.7 A system based on timed actor semantics. .............................. 71

4.1 A virtual machine supporting timed actor semantics. ................. 74
4.2 $A\text{Cube}$ definition for $b \in \text{Buffer}$. .............................. 79
4.3 Dispatcher, a finite state machine model of dispatcher behaviour. ............ 82
5.1 An abstract syntax of COOL specifications, $p \in \text{Specification}$. 103
5.2 An abstract syntax of COOL expressions, $e \in \text{Exp}$. 114
6.1 Steps in validating a COOL specification. 129
6.2 An abstract syntax of checkable properties, $\phi \in \text{Property}$. 131
6.3 A message sequence chart illustrating competition for a distributed binary lock with timeout. 134
6.4 Activations local to actor $a$. 136
6.5 An interpretation of timed intervals; all timed intervals are local to actor $a$. 137
6.6 A message sequence chart illustrating the interaction between a buffer $b \in \text{Buffer}$, a producer $p \in \text{Producer}$, and a consumer $c \in \text{Consumer}$ in a Producer-Consumer simulation. 139
6.7 A message sequence chart illustrating the interaction between a sender $s \in \text{Sender}$ and a receiver $r \in \text{Receiver}$ in alternating bit protocol. 141
6.8 The structure of a COOL event log. 144
6.9 Steps in the analysis of a trace database. 152
6.10 Running time versus round-trip time in application DABP. 159
7.1 Application DABP—assigning actors to nodes. 165
7.2 Message sequence charts illustrating the interaction between a sender $s \in \text{Sender}$ and a receiver $r \in \text{Receiver}$ in ABP. 167
7.3 Alternating bit protocol—changes in frame counters $\text{nfts}(\text{Sender})$ and $\text{fe}(\text{Receiver})$. 168
7.4 A message sequence chart illustrating the effect of incorrectly setting the duration of the acknowledgement timer in ABP. 170
7.5 Application DLOCK, message sequence chart illustrating a successful service access. 174
7.6 Application DLOCK, message sequence chart illustrating an unsuccessful service access. 175
7.7 A configuration for testing a distributed lock protocol. 177
7.8 A distributed lock $d \in \text{DLock}$. 180
7.9 A lock coordinator $l \in \text{Locker}$. 182
7.10 Message sequence charts illustrating the effect of lost messages on $l \in \text{Locker}$ and $d \in \text{DLock}$.

A.1 A typical message sequence chart.

A.2 Actor modelled as a timed communicating finite state machine.
## List of Listings

1.1 Specification for $m \in \text{Token}$. ................................................................. 13

2.1 Actor definition User. ................................................................. 43

2.2 Actor definition Vend. ................................................................. 44

4.1 Actor template for class Buffer. ................................................................. 80

6.1 Instrumenting message handlers `get` and `put`. ................................. 150

6.2 Algorithm `FindMonitorable`—finding monitorable communication events; and Algorithm `Instrument`—instrumenting a message handler. ................................. 151

B.1 `DspProcess`, `CheckTimers`, and `CheckInbound`—managing timers and inbound messages. ................................................................. 217

B.2 `CheckOutbound` and `CheckLocal`—managing inbound and local messages. ................................................................. 218

B.3 `vmprocess`—an operation for processing an intentions list. ................................................................. 219

B.4 `vmnew`—an operation for actor creation; and `vmtrigger`—an operation for timer creation. ................................................................. 220

B.5 `vmdiscard`—an operation for timer destruction; and `vmsend`, `vmforward`—operations for message transmission. ................................................................. 221

B.6 `vmbecomes`, `vmabort`, and `vmterminate`—the kernel commitment operations. ................................................................. 222

D.1 COOL specification for a vending machine—$v \in \text{Vend}$, $u \in \text{User}$, and $m \in \text{Main}$. ................................................................. 228

D.2 Application `DBUFF`—specification for $b \in \text{Buffer}$. ................................................................. 229

D.3 Application `DBUFF`—specification for $p \in \text{Producer}$, $c \in \text{Consumer}$, and $m \in \text{Main}$. ................................................................. 230
D.4  Application DABP—specification for $s \in \text{Sender}$, and $r \in \text{Receiver}$. . . . . . . . . 231
D.5  Application DABP—specification for $r \in \text{Receiver}$ (continued) and $u \in \text{User}$. . . 232
D.6  Application DABP—specification for $m \in \text{Main}^*$, constants, and function definitions. 233
D.7  Application DLOCK—specification for $d \in \text{DLock}$. . . . . . . . . . . . . . . . . . . . . . . . . . . 234
D.8  Application DLOCK—specification for $l \in \text{Locker}$. . . . . . . . . . . . . . . . . . . . . . . . . . . 235
D.9  Application DLOCK—specification for $u \in \text{User}$. . . . . . . . . . . . . . . . . . . . . . . . . . . 236
D.10 Application DLOCK—specification for $m \in \text{Main}^*$. . . . . . . . . . . . . . . . . . . . . . . . . . . 237
Acknowledgements

I wish to thank my supervisor, Dr. M. Cheng, for his advise and encouragement while conducting my research and preparing this dissertation. Dr. M. Cheng suggested this research project, and I am grateful for his guidance and support as the research progressed.

I would like to thank the external examiner, Dr. W. Older, for his constructive criticism of my dissertation and for his advise on how to clarify the timed actor semantics presented in Chapters 2 and 3. I am grateful to my examining committee, Dr. M. Cheng, Dr. G. C. Shoja, Dr. B. Kapron, Dr. I. Traoré, and Dr. W. Older, for agreeing to review my dissertation and for their suggestions for improving its presentation.

I would like to thank S. Calin Somosan and Anthony J. Howe for agreeing to use an early release of our COOL compiler to pursue their own research into a variety of distributed protocols [2, 3]. Many of their suggestions were incorporated into later releases of the compiler. Many graduate students, including Paul Wierenga, Dennis Zhou, Mike Zastre, and Steven Shen, provided useful feedback on a variety of issues during the development of my dissertation. I have also benefited greatly from the diverse approaches to research suggested by my course instructors during my doctoral program at the University of Victoria.

Financial support for this research was provided by Nortel Networks (Brampton, Ontario); and by the MITACS¹ Network of Centres of Excellence (Simon Fraser University, Burnaby, British Columbia).

Finally, I would like to thank my wife, Pat, for her encouragement, companionship and tolerance; and for providing me with a sanctuary in which to work and write.

¹Mathematics of Information Technology and Complex Systems
Chapter 1

Introduction

A distributed system is a collection of nodes connected by a shared communications network. Processes on these nodes cooperate to achieve common goals. These goals are often associated with global invariants which are maintained by the coordinated actions of the distributed processes.

Distribution adds complexity to distributed application development, in part due to:

- **Loss of a common clock**—In distributed systems there is a loss of the common clock found in centralized systems; requiring nodes to run asynchronously with respect to one another.

- **Unreliable communication**—In distributed systems the nodes are connected by a communications network and interprocess communication (IPC) is achieved either by synchronous or asynchronous message passing. This loose coupling of the nodes generally means that communications are subject to corruption, loss, and unbounded delay.

- **Possibility of partial failure**—In distributed systems, nodes or processes on a node are subject to failure.

- **Requirement for coordination**—Distributed applications often require a coordination\(^1\) strategy to guarantee that communication events occur in a desired order [5].

\(^1\)Coordination is the additional processing performed when multiple distributed processes pursue goals that a single process pursuing the same goals would not perform [4].
The added complexity of distribution makes it difficult to implement and validate distributed applications. This difficulty arises, in part, from the difficulty of abstracting distributed applications into a formal model where the correctness of an implementation may depend on the timing or the causal ordering of communication events.

The purpose of this dissertation is to describe our approach for modelling and implementing distributed applications:

- We develop a formal actor model \[6, 1, 5\] of distributed systems in which message processing is atomic, and actors\(^2\) create and manage timers. We refer to our model as the Timed Actor model, and its formal semantics as a timed actor semantics.

- We describe a virtual machine based on our timed actor semantics which can accommodate a timed actor language.

- We develop a timed actor language for specifying distributed applications. Our timed actor language adheres to our timed actor semantics.

- We develop a property driven technique for specifying, monitoring, and analyzing the performance and coordination properties of actors.

- We demonstrate the usefulness of our approach by applying it to a number of case studies.

Our approach is an attempt to unify the theory of distributed systems and the practice of implementing distributed applications into a simple framework for experimenting with distributed applications.

1.1 An Approach for Implementing Distributed Systems

In the Actor model \[1\] each node in a distributed system consists of a processor and a local actor configuration \(\langle K, X \rangle\) which consists of a set of actors \(K\), and a sequence of undelivered messages \(X\).

\(^2\)An actor is an active object with a private local state and its own thread of control.
System def = Node₁ | Node₂ | ⋅⋅⋅ | Nodeᵢ | ⋅⋅⋅ | Nodeₖ (1.1)

Nodeᵢ def = Processorᵢ | ⟨⟨K, X⟩⟩ᵢ (1.2)

In the Timed Actor model each node is provided with a local master-clock T which drives a set of local timers J. Support for atomic message processing is provided by an intentions list I which provides a record of the actions performed by an executing actor, and a set of commitment operators which implement local checkpointing.

Nodeᵢ def = Processorᵢ | Tᵢ | Jᵢ | ⟨⟨K, J, X⟩⟩ᵢ (1.2′)

1.1.1 Modelling and Specification

Modelling distributed applications. Our modelling approach employs message sequence charts [7] to capture the interaction between actors in a distributed system (for example see Figure 1.5 on page 11), and communicating finite state machines [8, 9, 10] to describe the behaviour of individual actors (for example see Table 1.3 on page 10 and Figure 1.4 on page 10). In our Timed Actor model, actor behaviour is expressed using the operations summarized in Table 1.1.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new(a, C)</td>
<td>Create actor a using definition C.</td>
</tr>
<tr>
<td>trigger(t, m(e), i, oneshot)</td>
<td>Create oneshot timer t with duration i and timeout message m(e).</td>
</tr>
<tr>
<td>trigger(t, m(e), i, periodic)</td>
<td>Create periodic timer t with period i and timeout message m(e).</td>
</tr>
<tr>
<td>discard(t)</td>
<td>Discard timer t.</td>
</tr>
<tr>
<td>send(a, m(e))</td>
<td>Send message m(e) to actor a.</td>
</tr>
<tr>
<td>forward(a)</td>
<td>Forward activation message to actor a.</td>
</tr>
<tr>
<td>becomes(q)³</td>
<td>Commit, transition to conceptual state q.</td>
</tr>
<tr>
<td>terminate³</td>
<td>Discard self.</td>
</tr>
<tr>
<td>abort³</td>
<td>Abort message processing.</td>
</tr>
<tr>
<td>{x ← e}</td>
<td>Assign expression e to state variable x.</td>
</tr>
</tbody>
</table>
Why use an actor model as the basis of a formal model? By adopting an actor model as the basis for our formal model we are accepting an asynchronous model. This is important for two reasons. First, asynchronous actor models tend to be more expressive than synchronous process models (process algebras and calculi). For example, all actor models and their associated actor languages, provide support for three special operations: new—to create new actors, send—to support interactor communication, and become—to specify replacement behaviour. In the Timed Actor model additional operators are introduced to manage timers and to implement atomic message processing (see Table 1.1). These operations allow the design of actor languages which easily accommodate the semantics of the Timed Actor model. In the π-calculus the semantics of timers and atomic message processing are not easily defined. We conclude that the Timed Actor model is more expressive than the π-calculus; for the issues we are addressing.

Second, synchronous process models and many actor models [11, 1] treat run-time exceptions as undefined. Since our modelling approach includes the production of executable implementations, our Timed Actor model describes the handling of run-time exceptions. In the Timed Actor model exception handling is described in the semantics of message processing through the commitment operators, and through the handling of undeliverable and rejected messages; and in the semantics of timers which allow an actor to react to the passage of time.

Specifying distributed applications. COOL is a timed actor language for specifying the interfaces, behaviour, and coordination of actors which implement distributed applications. COOL provides features essential for an effective coordination language: timer management, the ability to control the degree of concurrency, and the ability to distribute a computation. However, COOL is not a general purpose programming language, and does not provide general data structuring mechanisms.

COOL shares the following features with other actor programming languages [11, 12, 1, 5]:

- a new operator to create new actor instances, a send operator supporting asynchronous message

---

3The commitment operators provide support for local checkpointing and determine the role an actor will assume when it accepts its next input message.

4Replacement behaviour determines the role an actor will assume when it accepts its next input message.
passing, a **becomes** operator to specify *replacement behaviour, implicit*\(^5\) message reception, and actor instances with a *completely encapsulated* local state.

Several features distinguish COOL from other actor programming languages: class definitions are structured like *communicating finite state machines*, actors create and manage timers, message processing is *atomic* employing a *local checkpointing mechanism*, and class definitions can specify a set of *checkable properties* to be monitored at run-time.

A COOL specification is divided into two parts (see Figure 1.1); the definition of message handlers (**M**) which describe how actors react to message input, and the definition of checkable properties (**Φ**) which describe what actor behaviour to monitor. In COOL specifications, actor behaviour is expressed using the operations summarized in Table 1.2. In Table 1.2 we indicate how message sequence charts (**MSC**), and communicating finite state machines (**CFSM**) contribute to the development of a specification.

![Figure 1.1: Modelling and specification.](image)

### 1.1.2 Implementation

The creation of an executable application (**E**) requires (see Figure 1.2):

- A COOL compiler—**cool2L** translates the definition of actor behaviour into *virtual machine* instructions (**M'**) for *implementation language L*. The implementation module (**M'**) contains a template for each actor class defined in **M**. Each class template models an asynchronous **CFSM**. The **cool2L** compiler *enforces* the *timed actor semantics* and instruments the implementation to allow run-time monitoring of selected checkable properties defined in the spec-

---

\(^5\)Message reception is explicit if the model (language) provides a *receive* operator. Our *timed actor semantics* provides a *recv* operator (see Section 3.5.1), while in our *timed actor language* message reception is *implicit*. 
ification (Φ). cool2L translates the definition of checkable properties into a set of canonical checkable properties (Φ′) required by the trace analyzer.

Table 1.2: Operations for expressing actor behaviour in COOL.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = new C</td>
<td>MSC, CFSM</td>
<td>Create actor a using class definition C.</td>
</tr>
<tr>
<td>t = trigger m(e) on i</td>
<td>MSC, CFSM</td>
<td>Create oneshot timer t with duration i and timeout message m(e).</td>
</tr>
<tr>
<td>t = trigger m(e) every i</td>
<td>MSC, CFSM</td>
<td>Create periodic timer t with period i and timeout message m(e).</td>
</tr>
<tr>
<td>discard t</td>
<td>MSC, CFSM</td>
<td>Discard timer t.</td>
</tr>
<tr>
<td>m(e) ⇒ a</td>
<td>MSC, CFSM</td>
<td>Send message m(e) to actor a.</td>
</tr>
<tr>
<td>⇒ a</td>
<td>MSC, CFSM</td>
<td>Forward activation message to actor a.</td>
</tr>
<tr>
<td>becomes q</td>
<td>CFSM</td>
<td>Commit, transition to conceptual state q.</td>
</tr>
<tr>
<td>terminate</td>
<td>MSC, CFSM</td>
<td>Discard self.</td>
</tr>
<tr>
<td>abort</td>
<td>CFSM</td>
<td>Abort message processing.</td>
</tr>
<tr>
<td>x = e</td>
<td>CFSM</td>
<td>Assign expression e to state variable x.</td>
</tr>
</tbody>
</table>

- A VM—A linkable version of a virtual machine (VM) for implementation language L which enforces the timed actor semantics.

- A native compiler—A compiler for implementation language L is required to compile M′ and the VM.

These tools allow executable applications to be created using the native development tools on a typical development platform. An executable application E is produced by compiling and linking (denoted by +) the implementation module M′ with a linkable version of the VM for language L.

\[ E = \text{compile}(M') + \text{compile}(\text{VM}) \]

This development strategy has been tested for C/C++; a cool2C translator is described in [13, 14], and a VM for C/C++ is described in [15].

---

6Canonical checkable properties contain the information needed by our trace analyzer to parse an event log.
1.1.3 Validation

Validation is the comparison of an implementation with its specification. In our approach we compare the run-time behaviour (observed behaviour) of an implementation generated by the cool2L compiler with the expected behaviour of an implementation as defined by the checkable properties of the specification. For an implementation $E$ and a set of checkable properties $\Phi = \{\phi_1, \phi_2, \cdots, \phi_n\}$ we write

$$E \models \Phi$$

(1.3)

to indicate that the implementation $E$ satisfies checkable properties $\Phi$. In our approach $\Phi$ is expressed in terms of communication events, the reception and transmission of messages; and describes the expected behaviour and performance of actors in an executable implementation.

COOL allows the definition of the following types of checkable properties:

- An enumeration constraint, $\phi \triangleq \#z$, is used to collect information on the frequency of communication events. A communication event ($z$) is either the arrival (denoted $?m$) or transmittal
(denoted \(!m\)) of message \(m\).

- A *timing constraint*, \(\phi \triangleq z_1 - z_2\), is used to measure the interval of time between two communication events \(z_1\) and \(z_2\).

- A *follows constraint*, \(\phi \triangleq z_1 \rightsquigarrow z_2 : n\), is used to determine if two causally related events, \(z_1\) and \(z_2\), occur within a specified period of time \(n\).

- A *message precondition*, \(\phi \triangleq m(v) \text{ assert} (\text{expr}(v,s))\), evaluates the assertion \(\text{expr}(v,s)\) each time message \(m\) is activated. The assertion may contain references to message parameters \((v)\) and state variables \((s)\).

We use a monitoring approach to validate specifications written in COOL. By *monitoring* we mean techniques developed to observe systems at run-time. Such observations can potentially detect behavioural (protocol) errors, detect timing errors, and validate an implemented system against its specification. Monitoring can also assist in the measurement of selected characteristics of a running system. These include the enumeration of events, measuring the interarrival and intertransmittal times of events, estimating the reliability of protocols, estimating the round trip times of actor communication, and estimating the response time between events.

![Figure 1.3: Executing and monitoring an implementation.](image-url)
Figure 1.3 illustrates the key steps required to conduct the validation of an implementation \((E)\). During execution, monitoring results are stored as an in-memory database at each of the nodes participating in a distributed application. At the end of execution, the VM at each node outputs the in-memory database to a local event log (observed behaviour). All event logs generated by a run of an application are merged into a single offline database. A trace analyzer is used to examine the offline database for violations to the checkable properties (expected behaviour).

1.2 Example: A Distributed Token

Modelling a token. In a distributed system it is common for service users to acquire a token before accessing a shared service. A \textit{mutex} is a simple structure for implementing tokens. After initialization a token can accept two kinds of requests:

- \texttt{acquire()}—to acquire the token and its associated service. The actor granted access to the service is referred to as the \textit{holder} of the token.

- \texttt{release()}—to surrender the token and its associated service. Only the holder of the token can release the service.

The message sequence chart in Figure 1.5 illustrates a protocol for implementing a distributed token. In this example two service users \(u_1\) and \(u_2\) (or more generally users \(u_1\) to \(u_n\)) compete for a single token \(m \in \text{Token}\) which protects service \(s \in \text{Service}\). \text{Token} and \text{Service} are examples of actor classes.

The key component in this protocol is the token \(m \in \text{Token}\). The behaviour of this component is described by communicating finite state machine \(G_{\text{Token}} = (Q, Q_f, q_0, \Sigma_{\text{Token}}, B)\) (Figure 1.4), where

\[
\begin{align*}
Q &= \{\text{Start, Released, Acquired}\} \text{ is the set of conceptual states for } m \in \text{Token}, \\
Q_f &= \{\} \text{ is the set of final states for } m \in \text{Token}, \\
q_0 &= \text{Start} \text{ is the initial state for } m \in \text{Token}, \\
\Sigma_{\text{Token}} &= \{\text{init, acquire, release}\} \text{ are the messages accepted by } m \in \text{Token}, \text{ and} \\
B &= \text{ is the transition relation for } m \in \text{Token} \text{ summarized in Table 1.3.}
\end{align*}
\]
Table 1.3: Transition relation for $m \in \text{Token}$.

<table>
<thead>
<tr>
<th>$B$</th>
<th>Current State</th>
<th>$\Sigma_{\text{Token}}$</th>
<th>Condition$^8$</th>
<th>Next State</th>
<th>Actors Created</th>
<th>Messages Transmitted</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_1$</td>
<td>Start</td>
<td>init()</td>
<td></td>
<td>Released</td>
<td>{s} [s.init()]</td>
<td></td>
</tr>
<tr>
<td>$b_2$</td>
<td>Released</td>
<td>acquire()</td>
<td></td>
<td>Acquired</td>
<td>{} [u.acquired(s)]</td>
<td></td>
</tr>
<tr>
<td>$b_3$</td>
<td>Released</td>
<td>release() last = *</td>
<td></td>
<td>Released</td>
<td>{} [\star.released_r()]</td>
<td></td>
</tr>
<tr>
<td>$b_4$</td>
<td>Released</td>
<td>release() last \neq *</td>
<td></td>
<td>Released</td>
<td>{} [\star.reject()]</td>
<td></td>
</tr>
<tr>
<td>$b_5$</td>
<td>Acquired</td>
<td>release() holder = *</td>
<td></td>
<td>Released</td>
<td>{} [u.released()]</td>
<td></td>
</tr>
<tr>
<td>$b_6$</td>
<td>Acquired</td>
<td>release() holder \neq *</td>
<td></td>
<td>Acquired</td>
<td>{} [\star.reject()]</td>
<td></td>
</tr>
<tr>
<td>$b_7$</td>
<td>Acquired</td>
<td>acquire() holder = *</td>
<td></td>
<td>Acquired</td>
<td>{} [u.acquired_r(s)]</td>
<td></td>
</tr>
<tr>
<td>$b_8$</td>
<td>Acquired</td>
<td>acquire() holder \neq *</td>
<td></td>
<td>Acquired</td>
<td>{} [\star.reject()]</td>
<td></td>
</tr>
<tr>
<td>$b_9$</td>
<td>Acquired</td>
<td>release() last = *</td>
<td></td>
<td>Acquired</td>
<td>{} [\star.released_r()]</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.4: Communicating finite state machine for $m \in \text{Token}$.

Each interaction in Figure 1.5 has a corresponding transition in Table 1.3 ($b_2$–$b_9$). Transition tables indicate how an actor reacts to message input. For example, immediately after initialization when a token accepts message acquire(), it grants access to a protected service $s$ by issuing message acquired(s) to requester $u$ (see row $b_2$ in Table 1.3).

The remaining interactions illustrated in Figure 1.5 are described below:

- Granting token $m$ to actor $u$ is idempotent. The holder of the token may issue any number

$^1$Conceptual states identify the possible modes of operation for an actor belonging to class C.

$^8$In the transition table for $m \in \text{Token}$; last and holder refer to state variables in Listing 1.1, and $\star$ refers to the sender of the current message.
of acquire() requests. For requests 2· · · n, token m responds with message acquired_r(s) (b7). This behaviour guards against lost acquire() responses.

- If a second actor u′ attempts to acquire a held token, the token responds with message reject() (b8).

Figure 1.5: A message sequence chart illustrating competition for a distributed token.

- The holder of token m may release m at any time. Token m acknowledges a release() request
with message released() \((b_5)\).

- Releasing token \(m\) is idempotent. While a token is free, the last holder of token \(m\) may repeat the release() request any number of times. For requests \(2 \cdots n\), token \(m\) responds with message released_r() \((b_3, b_9)\). This behaviour guards against lost released() responses. If \(u\) is not the last holder of the token, token \(m\) responds with message reject() \((b_4)\).

- If a second actor \(u'\) attempts to release a held token, the token responds with message reject() \((b_6)\).

**Specification for class Token.** Listing 1.1 on page 13 contains the COOL specification of class Token. The first part of this specification defines the message handlers. The structure of class Token is derived from the communicating finite state machine illustrated in Figure 1.4. The message handlers for requests acquire() and release() operate as suggested by the transition relation in Table 1.3. The internal state for instances of class Token is provided by state variables last and holder.

The second part of the specification (starting with label Satisfies) defines the checkable properties for class Token. Checkable properties describe the expected behaviour of actor instances belonging to class Token. We define three checkable properties.

Property

\[
\phi_1 \triangleq \text{acquire()} \text{ assert}(\text{current} == \text{Released} \&\& \text{holder} == \text{none}) \tag{1.4}
\]

verifies that the token is free (holder == none) in conceptual state Released. This assertion can fail if state variable holder is being managed incorrectly.

Property

\[
\phi_2 \triangleq 0 \leq \#!\text{acquired} - \#!\text{released} \leq 1 \tag{1.5}
\]

is the mutex invariant; at any instant the number of successful acquire() requests (denoted \#!acquired) minus the number of successful release() requests (denoted \#!released) must be either 0 or 1.

One drawback with the definition of class Token is that once an actor acquires token \(m\), it cannot be compelled to release it. This situation can lead to starvation among the actors competing for token \(m\).
Listing 1.1: Specification for \( m \in \text{Token} \).

```cpp
actor class Token {
  Service s;
  actor holder, last;

Start:
  msg init () {
    holder = last = none;
    s = new Service;
    init () \Rightarrow s;
    becomes Released;
  }

Released:
  msg acquire () {
    holder = *;
    acquired (s) \Rightarrow holder;
    becomes Acquired;
  }

  msg release () {
    if (last == *) {
      released_r () \Rightarrow *;
    }
    else {
      reject () \Rightarrow *;
    }
  }

Acquired:
  msg release () {
    if (holder == *) {
      released () \Rightarrow holder;
      last = holder;
      holder = none;
      becomes Released;
    }
    else if (last == *) {
      released_r () \Rightarrow *;
    }
    else {
      reject () \Rightarrow *;
    }
  }

Satisfies:

\[ \phi_1 \triangleq \text{acquire()} \\text{assert(current == Released \&\& holder == none)} \]
\[ \phi_2 \triangleq 0 \leq \#!\text{acquired} - \#!\text{released} \leq 1 \]
\[ \phi_3 \triangleq !\text{acquired} \rightarrow !\text{released} : 30 \text{ msec} \]
```
One way to avoid this problem would be to associate a timer with token $m$. Once acquired, the token would grant access to service $s$ for a fixed period of time, say $n$ milliseconds. The token is released automatically if the holder fails to release token $m$ within $n$ milliseconds. Before we implement such a feature we use property

$$\phi_3 \triangleq !\text{acquired} \rightarrow !\text{released} : 30 \text{ msec} \quad (1.6)$$

to determine how the token is presently being used. Property 1.6 requests the trace analyzer to compute the mean and jitter of the token holding times. This information could then be used to determine the duration of the timer associated with the token.

**Validating class Token.** During translation the cool2L compiler determines that message handlers acquire() and release() need to be instrumented to monitor the defined checkable properties. cool2L generates an online monitor to record the following events:

Receive Events $\ ? \text{acquire, ?release}$
Transmission Events $\ !\text{acquired, !released}$
Assertion Events $\text{assert}_t(\phi_1)$

At run-time the online monitor logs the following traces

$$T_1 = \ ?\text{acquire} [\cdot\text{assert}_t(\phi_1)][\cdot!\text{acquired}] \quad (1.7)$$
$$T_2 = \ ?\text{release} [\cdot!\text{released}] \quad (1.8)$$

Trace $T_1$ is recorded each time message handler acquire() is activated. Assertion event $\text{assert}_t(\phi_1)$ is recorded only when Property 1.4 evaluates to false. Transmission event $!\text{acquired}$ occurs whenever token $m$ is granted. Trace $T_2$ is recorded each time message handler release() is activated. Transmission event $!\text{release}$ occurs whenever token $m$ is successfully released. These traces describe the observed behaviour of token $m$.

When the event log prepared by the online monitor is processed by our trace analyzer we can determine if the observed behaviour of $m \in \text{Token}$ is equivalent to the expected behaviour defined in the specification.
To validate Property 1.4 the trace analyzer needs to count the number of \( \text{assert}_f(\phi_1) \) events (denoted \(#\text{assert}_f(\phi_1)\)#) in the event log. If \(#\text{assert}_f(\phi_1) > 0\)# our implementation of class \text{Token} is not correct.

To validate Property 1.5 the trace analyzer enumerates events \(!\text{acquired}\) (denoted \(#!\text{acquired}\)#) and \(!\text{released}\) (denoted \(#!\text{release}\)#), evaluating the expression \((#!\text{acquired} - #!\text{acquired})\) each time one of these events is encountered. This evaluation must be either 0 or 1. If the mutex invariant does not hold our implementation of class \text{Token} is not correct.

To validate Property 1.6 the trace analyzer extracts timing information associated with every \(?\text{acquire}\) (stored in \(t_1\)) and \(?\text{release}\) (stored in \(t_2\)) event. When the trace analyzer encounters event \(!\text{released}\) after event \(!\text{acquired}\) has been seen, the interval between to two events, \(n = t_2 - t_1\), is computed and accumulated. If \(n > 30\) milliseconds, the trace analyzer reports a violation. When processing is complete the trace analyzer computes the mean and jitter of the token holding times from the accumulated results.

### 1.3 Related Work

#### 1.3.1 Coordination Models and Languages

Actor models and their associated languages are part of a larger family of coordination models and languages. Programming a distributed application can be treated as the union of two distinct activities: *programming the computation part*—which defines the objects responsible for computation, and *programming the coordination part*—which defines the objects responsible for coordinating the communications of objects involved in the computation [16].

A coordination model is expressed in terms of the triple \((E,M,L)\) where \(E\) is the set of coordinateable objects; \(M\) is the coordination media, the media which enables communication between the objects in \(E\); and \(L\) are the coordination laws, the actions performed by \(E\) to achieve coordination [17]. A coordination language is the linguistic embodiment of a coordination model [4]. Coordination languages provide a number of features, including: a facility for creating and terminating objects, a facility for communications, and a facility for distributing a computation.
Coordination models and languages are categorized as either data-driven or control-driven [4]. In data-driven models [4] processes wishing to communicate with each other use special coordination operators to read and write data in a shared medium. Communication in data-driven models often involves a broadcast facility. In data-driven languages coordination operators are generally added to a host language, and coordinators are specified using a mixture of coordination and computation code. Data-driven applications are often used to parallelize computation.

In control-driven models [4] processes communicate in a point-to-point fashion by means of a well-defined public interface. A control-driven model evolves by responding to events generated by transmission and reception activity. In control-driven languages the coordination and computation part of an application can be specified in a common language, and there is an almost complete separation of coordination and computation code. Control-driven applications are commonly used to model systems.

The Actor model [1] is described in [4] as a data-driven model in which actors are the coordination entities (E), a point-to-point communication network acts as the coordination medium (S), and the special operations new, send, and becomes form the basis of the coordination laws. The ActorSpace model [18] extends the point-to-point communication network in [1] by providing a broadcast operation.

1.3.2 The Actor Model

The Actor model [6] has long been recognized as a useful framework for studying distributed object-based systems [6, 1, 5]. The Actor model unifies the notions of objects and concurrency relying solely on asynchronous message passing as the means of communication between objects. The Timed Actor model, the model proposed here, extends the actor model of Agha [1] by providing three types of first-class objects; actors, timers, and messages; and by allowing atomic message processing. A comparison of our Timed Actor model and language with of the models and languages presented in [1, 5] is found in Table 1.4 on page 18. The important differences include:

- **Configurations**—In all models, configurations of the form \( \langle \mathcal{K}, \mathcal{X} \rangle \) are used to describe the system under study; where \( \mathcal{K} \) describes the actors present in the system, and \( \mathcal{X} \) holds a se-
quence of pending messages. A configuration in [5] takes the form \( \langle \mathcal{K}, S, X \rangle \), where \( S \) describes a set of synchronizer objects for specifying coordination constraints. Our Timed Actor model manages a set of timers \( \mathcal{J} \) resulting in configurations of the form \( \langle \mathcal{K}, \mathcal{J}, X \rangle \).

- **Actor class structure**—In [5] and our timed actor language a class definition facility is used to describe the local state of an actor, and its message interface. In our Timed Actor model message definitions are organized into blocks associated with named conceptual states. In our timed actor language a message may be defined in more than one message block. In [1] messages are identified by a tag field in each communication.

- **Threads of control**—In [5] and our Timed Actor model, actors have a single thread of control and concurrency within an actor is prohibited. Agha [1] allows multiple threads of control.

- **Message processing**—Message processing in [1, 19] has an interleaving semantics and atomicity exists only at the level of the special operations. This behaviour results from the semantics of the become replacement operator. Most treatments of actor semantics since [1] use the ready operation to implement replacement behaviour [5, 20, 21, 22]. The ready operation has an atomic semantic, that is, input messages are processed without preemption until a ready operation is executed.

  Our Timed Actor model utilizes a local checkpointing mechanism; input messages are processed atomically, however, the message handler for each received message either runs to completion, or it aborts discarding the current input message.

- **Message delivery**—There is considerable discussion [1, 5, 19] about the nature of the message delivery system in actor models. In [1, 5] message delivery is provided to mailboxes keyed to the globally unique identities of actors. The principal assumption in these models is that the underlying communications medium is reliable, so message delivery is guaranteed and fair; however, message ordering is not preserved.
Table 1.4: General properties of actor models.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Properties of (\pi)-agents and actors</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>threads of control</td>
<td>single</td>
<td>multiple</td>
<td>single</td>
<td>single</td>
</tr>
<tr>
<td>local state</td>
<td>absent</td>
<td>present</td>
<td>named</td>
<td>named, message blocks</td>
</tr>
<tr>
<td>message handlers</td>
<td>absent</td>
<td>tagged message structure</td>
<td>named</td>
<td>message guards and implicit message disabling</td>
</tr>
<tr>
<td>activation constraints</td>
<td>absent</td>
<td>absent</td>
<td>explicit disable, trigger and atomic constraints</td>
<td>atomic processing (indivisible)</td>
</tr>
<tr>
<td>message processing</td>
<td>interleaving semantics</td>
<td>interleaving semantics</td>
<td>atomic processing (indivisible)</td>
<td>atomic processing (local checkpointing)</td>
</tr>
<tr>
<td>specialized objects</td>
<td>absent</td>
<td>absent</td>
<td>synchronizer (S)</td>
<td>timer (J)</td>
</tr>
<tr>
<td>configurations</td>
<td>process expressions</td>
<td>({\mathcal{K}, \mathcal{X}})</td>
<td>({\mathcal{K}, S, \mathcal{X}})</td>
<td>({\mathcal{K}, J, \mathcal{X}})</td>
</tr>
</tbody>
</table>

**Properties of the message delivery system**

<table>
<thead>
<tr>
<th>organization</th>
<th>bag</th>
<th>mailbox</th>
<th>mailbox</th>
<th>partitioned FIFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>reliability</td>
<td>not specified</td>
<td>delivery guaranteed</td>
<td>delivery guaranteed</td>
<td>intranodal: reliable</td>
</tr>
<tr>
<td>ordering</td>
<td>not specified</td>
<td>not preserved</td>
<td>not preserved</td>
<td>internodal: unreliable</td>
</tr>
<tr>
<td>fairness</td>
<td>not fair</td>
<td>Mail delivery is guaranteed. System is fair because messages are eventually delivered.</td>
<td>Mail delivery is guaranteed. System is fair because messages are eventually delivered.</td>
<td>Messages are dispatched in FCFS fashion. Message is dropped if target actor has terminated. Message is dropped if target actor rejects message. Every message (\alpha \in \mathcal{X}) will eventually be processed.</td>
</tr>
</tbody>
</table>
In our Timed Actor model the sequence of pending messages $X$ at each node is partitioned into three sequences (see Figure 4.1 on page 74): $X_L$ is provided to support message delivery within a local configuration; and $X_I$ (inbound messages) and $X_O$ (outbound messages) are provided to support message delivery within a global configuration. This results in two types of delivery:

- intranodal delivery—which utilizes only $X_L$, is reliable and order preserving, and
- internodal delivery—which utilizes $X_I$ and $X_O$, is unreliable and not order preserving.

In our Timed Actor model pending messages are selected for dispatch in a first-come first-served (FCFS) fashion from either sequence $X_L$ or $X_I$. If a target actor has terminated, the message is discarded. If a target actor rejects the message (fails an input guard), the message is discarded. Since every message $\alpha \in X$ will eventually be processed, we characterize our message delivery system as fair.

Table 1.4 also includes a comparison between actors and $\pi$-agents [23, 24, 19]. $\pi$-agents differ from actors in the following ways: $\pi$-agents are stateless, $\pi$-agents employ an explicit input prefix (actor message reception is implicit), $\pi$-agent message delivery is not fair, and $\pi$-agent systems are unstructured (actor configurations are structured).

### 1.3.3 Actor Models with Activation Constraints

Our timed actor language supports activation constraints explicitly; using message guards to ensure that input messages are accepted only from specified sources, or under specified conditions; and implicitly, through the enabling and disabling of message handlers according to the semantics of the becomes commitment operator. In the following sections we describe some proposed models for coordinating groups of actors.

#### 1.3.3.1 Synchronizers

In [5] Frølund introduces a semantics for synchronizers. A synchronizer is a specialized object which encapsulates a group of actors. Synchronizers filter the messages exchanged by its com-
ponent actors, dispatching the messages according to user specified \textit{constraints}. Three general categories of constraints are described. \textit{Disable constraints} delay the dispatch of a message while a user-defined boolean expression evaluates to false. A \textit{trigger constraint}\footnote{Triggers in [5] execute arbitrary actions when specified messages are dispatched. In our \textit{Timed Actor model} triggers are timers.} monitors the messages dispatched to actor \(a\), executing a user-defined action each time message \(m\) is accepted by actor \(a\). \textit{Atomic constraints} prevent specific messages from being dispatched individually, requiring sets of messages to be dispatched in a single atomic action.

The actor model in [5] supports two granularities of atomicity; atomicity during message processing, and atomicity during message dispatch. Synchronizers address the issue of the ordering of messages; however, messages are subject to unbounded delay.

\subsection*{1.3.3.2 The Semantics of Real-Time Constraints}

A foundation for the semantics of actor-based real-time languages is presented in [25]. The model supports real-time constraints associated with the execution of message handlers, and the transmission of messages. In an actor class definition the header of each message handler must specify the worst case execution latency for message processing. For example, the definition of message handler \texttt{put()} in a bounded buffer implementation could be specified as \texttt{msg put(int x)<n>}, where \(n\) specifies the worst case execution latency.

In an untimed actor model a message transmission is specified by \texttt{send(a,m(e))} where \(a\) identifies a target actor and \(m(e)\) is a communication. In a timed actor model message transmission is specified by \texttt{send(a,m(e)) < r;d >}, where \(r\) is the release time for message \(m\) (the earliest time message \(m\) can be dispatched), and \(d\) is the deadline (the latest time message \(m\) can be invoked).

The authors show how the operational semantics of an untimed actor model can be extended to incorporate real-time constraints. The semantics requires at least one \textit{clock} for each pending message in a global configuration.
1.3.3.3 Actor-Based Real-Time Coordination Languages

Several authors [21, 22, 26] have combined the semantics of synchronizers and real-time constraints to describe specification languages for actor-based real-time systems. Access to a unique global reference time is a common requirement for these proposals. In [21, 27] globally synchronized clocks [28, 29] are proposed as a global time reference.

RTSynchronizers [22] extend the synchronizer model of [5] by adding real-time constraints and real-time triggers. Real-time constraints take one of two general forms; \( m_1 \Rightarrow m_2 < n \), which reads “after message \( m_1 \) has been dispatched the dispatch of message \( m_2 \) must occur within \( n \) units of time”; or \( m_1 \Rightarrow m_2 > n \), which reads “after message \( m_1 \) has been dispatched the dispatch of message \( m_2 \) must not occur for \( n \) units of time”. Real-time triggers take the form \( m : x = e \), which reads “when message \( m \) is dispatched local state variable \( x \) can be assigned expression \( e \)”. The authors demonstrate how an operational semantics of the basic actor model [1] can be extended to incorporate real-time constraints and triggers. They provide no discussion of the resources necessary to implement this model.

ATC [26] extends the synchronizer model of [5] by adding temporal constraints. In this model there are two types of constraints, and two special operations: \texttt{pasconst} \( c(a_1,a_2,\ldots,a_n) \) instantiates a passive constraint named \( c \) to coordinate actors \( a_1,a_2,\ldots,a_n \); and \texttt{actconst} \( c(a_1,a_2,\ldots,a_m) \) instantiates an active constraint named \( c \) to coordinate actors \( a_1,a_2,\ldots,a_m \). Passive constraint definitions take the form \( c : m_1 \texttt{watchdog}[d_1,d_2] m_2 \), which reads “for constraint \( c \), if message \( m_1 \) is not dispatched in the interval \([d_1,d_2]\) message \( m_2 \) will be dispatched”. Active constraint definitions take the form \( c : m \downarrow[d_1,d_2] \), which reads “for constraint \( c \), message \( m \) must be dispatched in the interval \([d_1,d_2]\) or the system will halt”. In both passive and active constraints the interval \([d_1,d_2]\) is implemented as \([t + d_1,t + d_2]\), where \( t \) is the time constraint \( c \) is instantiated.

1.3.4 COOL Case Studies

Constraint propagation protocol. In [2] Somosan presents a distributed architecture for solving the constraint propagation problem. In this architecture COOL is used to coordinate a set of actors;
each actor performing a specific role in the solution of an interval constraint problem. A constraint system is partitioned into a set of subsystems which are solved in parallel by a group of Worker actors. Each worker solves an assigned subsystem by continually exchanging information about its partial solution with other workers.

Adaptive client migration protocol. In [3] Howe designs and integrates an adaptive client migration protocol into a continuous data delivery network. Using this protocol, clients experiencing degradation of a data stream from a server have the ability to migrate to another server in order to improve their quality of service. The adaptive client migration protocol improves the resiliency of clients. A generic continuous delivery network was implemented in COOL. This continuous delivery system utilizes the proposed adaptive client migration protocol. Experiments validate the key properties (distributed initiation, idempotency, and resiliency) of the protocol.

1.4 Contributions

The research described in this dissertation extends the basic actor model [6, 1] by allowing the message interface of an actor to be structured as a communicating finite state machine, by allowing actors to manage timers, and by allowing atomic message processing with local checkpointing. We know of no other formal actor model and associated actor language providing this type of computation. The provision of these features in our Timed Actor model facilitates the modelling of actor coordination strategies.

Our operational semantics for the Timed Actor model, referred to as the timed actor semantics and represented by the core language ACube, is provided in sufficient detail to allow us to design a timed actor language called COOL [30, 14], and to describe a virtual machine supporting a timed actor semantics.

The COOL language facilitates the simultaneous specification of actor behaviour (M), and mon- itorable checkable properties (Φ). A specification (M,Φ) describes the expected behaviour of an actor class. The specification of actor behaviour is compiled into virtual machine instructions and
executed on our virtual machine for a **timed actor language**. The specification of checkable properties can be compiled into an **online monitor**. At run-time the online monitor records the **observed behaviour** of each instance of a monitorable class. Our **offline trace analyzer** validates an implementation by comparing the observed behaviour of monitored actors with their expected behaviour.

The **COOL compiler and virtual machine** described in [13, 14, 15]; and the trace analyzer described in Sections 6.3–6.4 have allowed us to model, specify, implement and validate several case studies. These case studies include a version of **alternating bit protocol**, a version of **sliding window protocol** [13], a **distributed locking protocol**, and a simulation of a **GSM handover protocol** [13].

Our approach strikes a balance between the theoretical aspects of distributed systems (Chapters 2, 3, and 4), and the practice of implementing experimental distributed applications (Chapters 6 and 7). **COOL** (Chapter 5) acts as a **bridge** between the formal and pragmatic aspects (see Figure 1.6).
1.5 Organization of the Dissertation

In Chapter 2, we introduce the *Timed Actor model* and describe how it can be used to structure a distributed system. An informal semantics of the *Timed Actor model* is presented in the form of a simple *timed actor language*. This informal semantics identifies the programming language constructs necessary to define a *timed actor language*. In Appendix A.1 we provide a brief description of message sequence charts, and in Appendix A.2 we describe the structure of a communicating finite state machine.

In Chapter 3, we describe an *operational semantics* for a programming language that adheres to our *Timed Actor model*. We refer to this semantics as a *timed actor semantics*. Our treatment focuses on the semantics of commitment (*becomes, terminate, abort*), the semantics of asynchronous communication (*send, forward*), the semantics of actor creation (*new*), and the semantics of timers (*trigger, discard*).

In Chapter 4, we describe some of the characteristics of a kernel (VM) to accommodate a *timed actor language*. We examine the model of actor execution supported by our kernel, describe the unit of actor execution (the *actor template*), and describe the implementation of our kernel. In Appendix B we provide a summary of the key syntactic domains used to describe the kernel, and the pseudocode for a kernel supporting a *timed actor language*.

In Chapter 5, we describe COOL, a *timed actor language* for specifying the interfaces, behaviour, and coordination of actors which implement distributed applications. COOL is an *object-based* language which allows the declaration of timers, supports a *local checkpointing mechanism*, allows the definition of *drop probabilities*, and allows the definition of *checkable properties* which can be monitored at run-time.

In COOL, the performance and coordination properties of actors may be specified and subsequently compiled into an *online monitor*. In Chapter 6, we describe how to specify checkable properties in COOL specifications, and describe how a COOL compiler instruments an implementation to allow the monitoring of checkable properties at run-time. At run-time the implementation generates a log of *selected actor activity* amenable to *offline trace analysis*. Our trace analyzer identifies when
and where checkable properties are violated. Appendix C provides information on the structure of COOL event logs.

In Chapter 7, we describe our experience with modelling, specification, implementation and validation of two distributed protocols. In Section 7.2 we describe application DABP, an implementation of the alternating bit protocol (ABP); and in Section 7.3 we describe application DLOCK, a simulation of a distributed lock protocol. The specifications for these applications are provided in Appendix D.

In Chapter 8, we state our findings and suggest some future work.
Chapter 2

Actor Systems

The Actor model [1] has long been recognized as a useful abstraction for studying distributed object-based systems [20, 5, 19]. An actor is an active object with a private local state and a single thread of control. Actors communicate by sending messages and react by mapping input messages into behaviour. Processing an input message leads to three distinct types of actions: the creation of a finite set of actors, the generation of a finite sequence of output messages, and the execution of the become operator which finalizes message processing [1].

A distributed system is a collection of \( k \) nodes connected by a shared communications network. In the Actor model each node in a network consists of a processor and a local actor configuration \( \langle \mathcal{K}, \mathcal{X} \rangle \), which consists of a set of actors \( \mathcal{K} \), and a sequence of pending messages \( \mathcal{X} \). To simplify the discussion of configurations we focus on the global actor configuration \( \{\mathcal{K}, \mathcal{X}\} \) defined by Equation 2.3. Actors in a configuration cooperate to achieve common goals. These goals are often expressed as global invariants which are maintained by the coordinated activities of the actors.

\[
\begin{align*}
\text{System} & \overset{\text{def}}{=} \text{Node}_1 \mid \text{Node}_2 \mid \cdots \mid \text{Node}_i \mid \cdots \mid \text{Node}_k \\
\text{Node}_i & \overset{\text{def}}{=} \text{Processor}_i \mid \{\mathcal{K}, \mathcal{X}\}_i \\
\{\mathcal{K}, \mathcal{X}\} & \overset{\text{def}}{=} \bigcup_{i=1}^{k} \{\mathcal{K}, \mathcal{X}\}_i
\end{align*}
\]

In this chapter we describe an extension of the Actor model which accommodates a set of timers \( \mathcal{J} \), and facilitates the atomic processing of input messages. We refer to this model as the Timed
**Actor model.** The *Timed Actor model* is motivated by the theory of *timed automata* and *timed process algebras* [31, 32, 33, 34].

In our *Timed Actor model* each node is provided with an unresettable\(^1\) local master-clock \(T\), from which we derive a discrete periodic action \(\sqrt{}\) (pronounced *tick*) to mark the progress of time. Support for atomic message processing is provided by an *intentions list* \(J\) which provides a record of the actions performed by an executing actor, and a set of *commitment operators* which implement a local *checkpointing* mechanism.

In our *Timed Actor model* processing an input message leads to five distinct types of actions: the creation of a finite set of actors, the creation of a finite set of timers, the destruction of a finite set of timers, the generation of a finite sequence of output messages, and the execution of a *commitment operator* which finalizes message processing. During message processing the actions of an actor are recorded on an *intentions list* \(J\). If processing succeeds the actions on \(J\) are *committed*, updating the local actor configuration; otherwise, all actions are abandoned and the local actor configuration is *rolled back* to the point where the current input message was *accepted*. To simplify the discussion of configurations in the *Timed Actor model* we focus on the refined global actor configuration \(\langle\langle K, J, X \rangle\rangle\) defined by Equation 2.3'.

\[
\text{Node}_i \overset{\text{def}}{=} \text{Processor}_i \mid T_i \mid J_i \mid \langle\langle K, J, X \rangle\rangle_i \\
\langle\langle K, J, X \rangle\rangle \overset{\text{def}}{=} \bigcup_{i=1}^{k} \langle\langle K, J, X \rangle\rangle_i
\]

**Outline.** In Sections 2.1 and 2.2 we describe the components of the *Timed Actor model*. In Section 2.3 we use a simple timed actor language to present an informal semantics of the *Timed Actor model*. In Section 2.4 we specify a vending machine configuration using STAL actor definitions.

\(^1\)\(T\) cannot be reset since this would affect the precision of the timers and confound the interpretation of timed checkable properties (see Section 6.2.2).
2.1 Basic Actor Model

2.1.1 Actor Instances

The set of all possible actor instances, \( K \), is given by \( K = A \times Q \times S \) where \( A \), with typical elements \( a, a_1, a_2, \ldots \), is the set of all possible actor identities; \( Q \), with typical elements \( q, q_0, q_1, \ldots \), is the set of all possible conceptual states; and \( S \), with typical elements \( s, s_0, s_1, \ldots \) is the set of all possible local states. We think of the conceptual states of an actor as different modes of operation, while the local state describes the internal representation of its computation. For example, an actor may have an internal buffer of arbitrary size as its local state, but conceptually its mode of operation may only depend on whether the buffer is Empty, Full, or PartFull.

Let \( a \in A \) be the identity of an actor, \( \{q_0, q_1, \ldots, q_k\} \subseteq Q \) represents the set of conceptual states actor \( a \) assumes during its lifetime, and \( \{s_0, s_1, \ldots, s_l\} \subseteq S \) represents the set of local states actor \( a \) assumes during its lifetime. An actor instance with identity \( a \) is denoted by

\[
K_a = (a, q_i, s_i)
\]  

(2.4)

where \( q_i \) is its current conceptual state, and \( s_i \) is its current local state.

The set of actor instances in a system, \( \mathcal{K} \subseteq K \), is a finite mapping from actor identity to conceptual state and local state. Actor instances are created using the \texttt{new} operator (see Section 2.3 on page 38). The creator of actor \( a \) is the owner of actor \( a \). The initial instance of a new actor with identity \( a \) is denoted \((a, q_0, s_0)\) where \( q_0 \) is its initial conceptual state, and \( s_0 \) is its initial local state.

The \texttt{new} operator generates globally unique actor identities. Actor instances may self-terminate using the \texttt{terminate} operator (see Section 2.3 on page 40). The terms actor, timed actor, and actor instance are equivalent.

2.1.2 Messages

In an actor system the reception of a message causes the activation of a target actor. A message is represented by a 4-tuple \((\star, a, \sigma, v)\) where \( \star, a \in A \) are the source and destination actor identities respectively, \( \sigma \in \Sigma \) is the message name, and \( v \in V \) is the message contents. The set of all possible
messages, $X$, is given by

$$X = A \times A \times \Sigma \times V$$  \hspace{1cm} (2.5)

where $A$ is the set of all possible actor identities; $\Sigma$, with typical elements $\sigma, \sigma_1, \sigma_2, \ldots$, is the set of all possible message names; and $V$, with typical elements $v, v_1, v_2, \ldots$, is the set of all possible message contents.

The pending messages at a node are organized as a sequence $X \in X^*$. Messages are created using the send operator $\Rightarrow$ (see Section 2.3 on page 38). If $\star$ is the identity of the source actor, $\sigma(v) \Rightarrow a$ generates message $(\star, a, \sigma, v)$; i.e., “message $\sigma$ with message contents $v$ is created by source actor $\star$ for delivery to target actor $a$”.

**Input set.** If actor $a$ is an instance of actor definition $D$ (see description of Equation 2.14 on page 36) then $\Sigma_a = \Sigma_D \subseteq \Sigma$, and $\Sigma_D$ is the set of input messages actor $a$ can receive.

**Message delivery.** In our Timed Actor model communication is asynchronous and point-to-point between actors, or between an actor and a timer. In our model a target actor may choose to drop an input message, when the input message is undefined, when the input message is disabled, or when the input message is incorrectly routed (see Section 2.2.2). In the implementation of a protocol, dropped input messages cannot be distinguished from a failed node or a failed communication. For these reasons, in the Timed Actor model message delivery is not reliable and not order preserving.

### 2.2 Timed Actor Model

#### 2.2.1 Timer Instances

The set of all possible timer instances, $J$, is given by $J = T \times Q \times S'$ where $T$, with typical elements $t, t_1, t_2, \ldots$, is the set of all possible timer identities; $Q$, with typical elements $q, q_0, q_1, \ldots$, is the set of all possible conceptual states; and $S' \subseteq S$, with typical elements $s, s_0, s_1, \ldots$ is the set of all possible local states. The local state of a timer is an element of $S'$

$$S' = X \times \mathbb{N} \times \{ \text{oneshot, periodic} \} \hspace{1cm} (2.6)$$
where $X$, with typical elements $\beta, \beta_1, \beta_2, \cdots$, is the set of all possible timeout messages; $\mathbb{N}$, with typical elements $n, n_0, n_1, \cdots$, is the set of all possible counter values; and $\{\text{oneshot, periodic}\}$ is the set of all possible timer types.

Let $t \in T$ be the identity of a timer, $\{q_0, q_1, \cdots, q_k\} \subseteq Q$ represents the set of conceptual states timer $t$ assumes during its lifetime, and $\{s_0, s_1, \cdots, s_l\} \subseteq S'$ represents the set of local states timer $t$ assumes during its lifetime. A timer instance with identity $t$ is denoted by

$$J_t = (t, q_i, (\beta_i, n_i, p_i))$$ (2.7)

where $q_i$ is its current conceptual state, $\beta_i$ is its current timeout message, $n_i$ is its current counter value, and $p_i$ the current timer type.

In our Timed Actor model timers are treated as specialized actors with a predefined local state. This allows timeout events to be represented by messages (see Section 2.1.2). The set of timer instances in a system, $\mathcal{J} \subseteq J$, is a finite mapping from timer identity to conceptual state and local state. $\mathcal{J}$ is partitioned into two sets; one for oneshot timers, and one for periodic timers, $\mathcal{J} = \mathcal{J}_{\text{oneshot}} \cup \mathcal{J}_{\text{periodic}}$.

Timer instances are created and activated using the trigger operator (see Section 2.3 on page 39). The creator of timer $t$ is the owner of timer $t$. The initial instance of a new timer with identity $t$ is denoted $(t, q_0, (\beta_0, n_0, p_0))$ where $q_0$ is its initial conceptual state, $\beta_0$ is its initial timeout message, $n_0$ is its initial counter value, and $p_0$ is its initial timer type. The trigger operator generates globally unique timer identities. A timer is destroyed by its owner using the discard operator (see Section 2.3 on page 39). The terms timer and timer instance are equivalent. Section 2.2.4 describes the behaviour of timers.

### 2.2.2 Actor Behaviour

Actors are computational agents which map input messages into a finite set of actor creation operations, a finite set of timer management operations, a finite set of output messages, and a commitment operation. The behaviour of an actor (in the sense of [11]) is an element of $B$

$$B = K \times X \longmapsto K \times \mathcal{P}(K) \times \mathcal{P}(J) \times X^* \times R$$ (2.8)
where \( K \) is the set of all possible actor instances, \( \mathcal{P}(K) \) is the set of all finite subsets of actor instances, \( \mathcal{P}(J) \) is the set of all finite subsets of timer instances, \( X^* \) is all possible sequences of output messages, and \( R \) is the set of all possible commitment operations (see Definition 2.1).

**Local checkpoints.** Each instance of actor behaviour has an antecedent of the form \((K_a, \alpha) = ((a,q_i,s_i),(\cdot,a',\sigma,v))\). When \( a = a' \) and \( \sigma \in \Sigma_a \), actor \( a \) accepts input message \( \alpha \) causing the generation of checkpoint \((\omega,q_i,s_i,J = [])\), where \( \omega \) is the activation time of actor \( a \), \( q_i \) is the current conceptual state of actor \( a \), \( s_i \) is a copy of the current local state \( s_i \) of actor \( a \); and \( J \) is the intentions list, initially empty. When \( a \neq a' \) or \( \sigma \notin \Sigma_a \), actor \( a \) rejects input message \( \alpha \). A rejected input message is discarded and treated as a lost message. Each antecedent has an image of the form \(((a,q_j,s_j),I,r_j)\) where \( s_j \) is the modified local state of actor \( a \), \( q_j \) is the next conceptual state of actor \( a \), \( I \) is the intentions list, and \( r_j \) is a commitment operation. An image describes the response of an actor to input message \( \alpha \).

During message processing, an intentions list \( J \) provides a record of the order in which new, trigger, discard, and send operations are performed by an actor \( a \). The projection \( J \uparrow \text{op} \) returns the sequence of operations of type \text{op} on intentions list \( J \). Every intentions list has four partitions

\[
K^+ = J \uparrow \text{new} \quad J^+ = J \uparrow \text{trigger} \quad J^- = J \uparrow \text{discard} \quad X^+ = J \uparrow \text{send}
\]  

(2.9)

where \( K^+ \) is the set of new actors created by \( a \), \( J^+ \) is the set of new timers created by \( a \), \( J^- \) is the set of active timers discarded by \( a \), and \( X^+ \) is the sequence of messages generated by \( a \).

**Commitment operations.** In our Timed Actor model, processing a message is an atomic operation; it either succeeds or aborts. When an actor \( a \) accepts a message \( \alpha \), it is required to finalize its message processing by executing one of the following commitment operations:

- **becomes** \( q \)—when it successfully processes \( \alpha \) and transitions to conceptual state \( q \),
- **terminate**—when it successfully processes \( \alpha \) and self-terminates, or
- **abort**—when it aborts processing \( \alpha \). When an actor aborts, all changes made during the processing of \( \alpha \) are undone, i.e.,
the conceptual state $q_i$ and the local state $s_i$ of actor $a$ are restored from its checkpoint $(\omega, q_i, s_c, J)$,

- all actors created are removed,
- all timers created are removed,
- all timers discarded are restored, and
- all generated messages are discarded.

The effect of the **abort** operation is to rollback the *configuration* to the state that existed when $\alpha$ was *accepted* and treat $\alpha$ as a *lost message*.

**Definition 2.1 (Commitment)** The set of all possible commitments, $R$, with typical elements $r, r_1, r_2, \ldots$, is given by

$$ R = \{ \text{abort, terminate} \} \cup \{ \text{becomes } q \mid q \in Q \} $$

where $Q$ is the set of all possible conceptual states.

### 2.2.3 Timed Actor Configurations

A system based on the behaviour of actors described by Equation 2.8 leads to a *timed configuration* composed of actor instances, timer instances and pending messages.

**Definition 2.2 (Timed Actor Configuration)** A timed configuration $C$ is the 3-tuple $\langle (\mathcal{K}, \mathcal{J}, \mathcal{X}) \rangle$ where $\mathcal{K}$ is the set of actor instances in the system, $\mathcal{J}$ is the set of timer instances in the system, and $\mathcal{X} \subseteq X^*$ is a finite sequence of messages.

Several functions are used to discover the condition of a timed configuration: $\text{actors}(\mathcal{K})$ returns the list of actor identities present in $\mathcal{K}$, $\text{timers}(\mathcal{J})$ returns the list of timer identities present in $\mathcal{J}$, and $\text{owner}(x)$ returns the identity of the owner of actor or timer $x$. $\text{owner}(x)$ returns $\text{none}$ if the owner of actor or timer $x$ has terminated. $\text{none}$ denotes a nonexistent actor. $\text{head}(\mathcal{X})$ returns the next message to be processed, and $\text{tail}(\mathcal{X})$ returns the tail of the pending message list. Any sequence of pending
messages $\mathcal{X}$ can be represented by $\operatorname{head}(\mathcal{X}) \bowtie \operatorname{tail}(\mathcal{X})$, where the operator $\bowtie$ specifies concatenation of sequences.

Let $\mu \in \{\alpha, \beta, \sqrt{\cdot}\}$ be an action which may cause a timed configuration to transition, where, $\alpha$ is the acceptance of an activation message, $\beta$ is the acceptance of a timeout message generated by a timer, and $\sqrt{\cdot}$ is the periodic tick of the master-clock. The timed configuration transition relation describes how a timed configuration evolves.

**Definition 2.3 (Timed Configuration Transition Relation)** Let $C_1$ and $C_2$ be two timed configurations of the form $C_i = \langle\langle K_i, J_i, X_i \rangle\rangle$. $C_1$ transitions to $C_2$, denoted $C_1 \xrightarrow{\mu} C_2$, if:

- **Case 1:** $\mu$ is an activation message $\alpha = \operatorname{head}(X_1) = (\star, a, \sigma, v)$ generated by actor $\star$ and accepted by an actor $a \in \operatorname{actors}(K_1)$;

- **Case 2:** $\mu$ is a timeout message $\beta = \operatorname{head}(X_1) = (t, a, \sigma, v)$ generated by timer $t$ and $\beta$ is accepted by an actor $a \in \operatorname{actors}(K_1)$ with $a = \operatorname{owner}(t)$;

- **Case 3:** $\mu = \sqrt{\cdot}$ and one or more timeout messages $\beta_1, \cdots, \beta_n$ are generated (adjusting $J_1$ for the removal of oneshot timers $J_{\text{oneshot}}$):

$$\langle\langle K_1, J_1, X_1 \rangle\rangle \xrightarrow{\sqrt{\cdot}} \langle\langle K_1, J_1 - J_{\text{oneshot}}, X_1 \bowtie [\langle \beta_1 \rangle, \cdots, \langle \beta_n \rangle] \rangle\rangle$$

- **Case 4:** $\mu = \sqrt{\cdot}$ and no timeout messages are generated ($J_1'$ indicates that accepting $\sqrt{\cdot}$ changes to the local state of all timers); or

$$\langle\langle K_1, J_1, X_1 \rangle\rangle \xrightarrow{\sqrt{\cdot}} \langle\langle K_1, J_1', X_1 \rangle\rangle$$

- **Otherwise,** $\mu$ is discarded.

$$\langle\langle K_1, J_1, \langle \mu \rangle \bowtie X_1 \rangle\rangle \xrightarrow{\mu} \langle\langle K_1, J_1, X_1 \rangle\rangle$$

The response of actor $a$ on accepting message $\mu' \in \{\alpha, \beta\}$ is described by

$$B((a, q_i, s_i), \mu') = ((a, q_j, s_j), K^+, J^+, J^-, X^+, r_j) \quad (2.10)$$
where $K^+, J^+, J^−$, and $X^+$ are the partitions of $I$ defined in Equation 2.9. The configuration update operator, $⊙$ (Equations 2.11–2.13), computes the next state of a configuration. In the transition $C \xrightarrow{\mu'} C' \uplus \mathcal{J}$, $C'$ is configuration $C$ with input message $\mu'$ removed from the list of pending messages, and operator $\uplus$ combines configuration $C'$ with the contents of intentions list $\mathcal{J}$.

More precisely, the current actor will process message $\mu'$, transforming configuration $C' = \langle \langle K_1, J_1, X_1 \rangle \rangle$ into configuration $C' \uplus \mathcal{J} = \langle \langle K_2, J_2, X_2 \rangle \rangle$ where:

$$\begin{align*}
\mathcal{K}_2 &= \begin{cases} 
\mathcal{K}_1 \cup K^+ : & r_j \text{ becomes} \\
\mathcal{K}_1 - \{K_a\} \cup K^+ : & r_j \text{ terminate} \\
\mathcal{K}_1 : & r_j \text{ abort}
\end{cases} \\
J_2 &= \begin{cases} 
J_1 - J^- \cup J^+ : & r_j \text{ becomes, terminate} \\
J_1 : & r_j \text{ abort}
\end{cases} \\
X_2 &= \begin{cases} 
tail(X_1) \cap X^+ : & r_j \text{ becomes, terminate} \\
tail(X_1) : & r_j \text{ abort}
\end{cases}
\end{align*}$$

(2.11) (2.12) (2.13)

Figure 2.1 illustrates the relationship between the configuration transition relation and the behaviour of an actor. Each dispatched message $\mu$ is either rejected or accepted by an actor $a$. Rejecting $\mu$ results in the discard of $\mu$ from $X_1$. Accepting $\mu$ leads to a response from actor $a$. This response determines how configuration $\langle \mathcal{K}_1, J_1, X_1 \rangle$ is updated to $\langle \mathcal{K}_2, J_2, X_2 \rangle$.

Figure 2.1: The relationship between actor behaviour and changes in a configuration.
2.2.4 Timer Behaviour

In the Timed Actor model, actor and timer instances are modelled as communicating finite state machines [10] (see Appendix A.2). The timers described in Section 2.2.1 are instances of \( G_{\text{Timer}} = (Q, Q_f, q_0, \Sigma_{\text{Timer}}, B) \) shown in Figure 2.2, where

\[
Q = \{\text{Start, Tick, 0}\}, \quad Q_f = \{0\}, \quad q_0 = \text{Start}, \quad \Sigma_{\text{Timer}} = \{\sqrt{\}} \cup \{\text{trigger, discard}\}.
\]

The transition relation for \( G_{\text{Timer}} \) is presented in Table 2.1. Columns \( q_{\text{current}} \) and \( q_{\text{next}} \) show the current and next conceptual state, column \( \Sigma_{\text{Timer}} \) lists the input actions and operations accepted by timers, column Output shows the message sequences generated by an instance of Timer, and \( R \) shows the commitment operations.

Table 2.1: Transition relation for timer \( t \in \text{Timer} \).

<table>
<thead>
<tr>
<th>( q_{\text{current}} )</th>
<th>( \Sigma_{\text{Timer}} )</th>
<th>Condition</th>
<th>( q_{\text{next}} )</th>
<th>Output</th>
<th>( R )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>trigger(( \beta ),n,p)</td>
<td></td>
<td>Tick</td>
<td>[]</td>
<td>becomes Tick</td>
</tr>
<tr>
<td>Tick</td>
<td>trigger(( \beta ),n,p)</td>
<td></td>
<td>Tick</td>
<td>[]</td>
<td>becomes Tick</td>
</tr>
<tr>
<td>Tick</td>
<td>discard(t)</td>
<td></td>
<td>0</td>
<td>[]</td>
<td>terminate</td>
</tr>
<tr>
<td>Tick ( \sqrt{} )</td>
<td>( p = \text{periodic} )</td>
<td></td>
<td>Tick ( [\beta] )</td>
<td>becomes Tick</td>
<td></td>
</tr>
<tr>
<td>Tick ( \sqrt{} )</td>
<td>( p = \text{oneshot} )</td>
<td></td>
<td>0 ( [\beta] )</td>
<td>terminate</td>
<td></td>
</tr>
</tbody>
</table>

The input set for timer instance \( t \in \text{Timer} \) includes the action \( \sqrt{\} \), and the timing operators trigger and discard:

- \( \sqrt{\} \)—This action is broadcast by the master-clock \( \mathbb{T} \) at a fixed period to update all timers in a local timed actor configuration. Timers decrement each time \( \sqrt{\} \) is accepted. When a timer expires (decrements to 0), a timeout message \( \beta \) is sent to the owner of the timer.

- trigger—The trigger operator creates and enables new timers.

- discard—The discard operator disables and removes a timer from the local timed actor configuration.
2.3 STAL—A Simple Timed Actor Language

In this section we describe the constructs necessary to define a simple timed actor language (STAL). STAL is introduced here to allow the description of the vending machine example in Section 2.4. A more comprehensive treatment of a timed actor language is presented in Chapter 5.

To be useful a timed actor language must be able to specify the desired behaviour of the actors which make up an actor system. We define the behaviour of an actor using a grammar for actor definitions:

\[
\begin{align*}
\text{def } & D(s, s', \cdots) K \text{ enddef} \\
K & ::= K' K'' | q_i : M | \text{Defaults} : M \\
M & ::= M' M'' | \text{msg } \sigma_j(v, v', \cdots) \{P_j\}
\end{align*}
\]

where \(D\) is the name of the actor definition, \((s, s', \cdots)\) is the local state accessible by an instance of \(D\), \(q_i \in Q\) is a conceptual state for actor definition \(D\), and \(\text{msg } \sigma_j(v, v', \cdots) \{P_j\}\) defines the message handler \(P_j\) associated with input message \(\sigma_j\). The contents of a message consists of a set of message parameters \(v, v', \cdots\). For the sake of simplicity, we avoid specifying any type information in the definition of local state variables and message parameters.

An actor definition provides a template for describing the behaviour of an actor. An actor definition is organized as a sequence of message blocks \((K)\). Each message block describes the

![Communicating finite state machine for \(t \in \text{Timer}\).](image-url)
messages an actor may accept in a specific conceptual state \((q_i)\) and the message handlers \((M)\) that are dispatched. We denote that an actor \(a\) is instantiated from actor definition \(D\) by \(a \in D\).

The actor definition for the vending machine in Section 2.4 takes the following form:

\[
\text{def } \text{Vend}(u,x) \\
\text{Start: } \\
\text{msg } \text{init}(u) \{ P_1 \} \\
\text{Select: } \\
\text{msg } \text{select}(y) \{ P_2 \} \ldots \\
\text{VM10: } \\
\text{msg } \text{coin}(y) \{ P_{j-1} \} \\
\text{Defaults: } \\
\text{msg } \text{restart}() \{ \text{becomes Select} \} \\
\text{enddef}
\]

This definition indicates that the local state of \(\text{Vend}\) consists of two instance variables \(u\) and \(x\). An actor definition can contain a special message block labelled \(\text{Defaults} \notin Q\) which defines message handlers accessible in all conceptual states.

To specify a message handler we require, at a minimum, operators to create new actors, manage timers, send messages, and commitment operators. Message handlers are constructed using the following grammar, where \(c^*\) denotes 0 or more control operations and \(r\) denotes a commitment operator.

\[
P \ ::= \ c^* ; r \\
P_t, P_f \ ::= \ P \mid c^* \\
c \ ::= \ x = e \mid \text{if } (e) \ P \mid \text{else } P_f \mid a = \text{new } D \mid f(e) \Rightarrow a \\
\text{t} = \text{trigger}(f(e), n, \text{oneshot|periodic}) \mid \text{discard } t \\
r \ ::= \ \text{becomes } q \mid \text{terminate} \mid \text{abort}
\]

Every actor has access to a read-only instance variable called \(\text{now}\), which is assigned the current value of the master-clock \(T\) each time the actor is activated (dispatched). That is, each time an actor begins processing an input message, the \(\text{now}\) instance variable is updated from the master-clocks value. Execution of a message handler is atomic, once initiated an actor will not be preempted until a commitment point is reached.
Sequence operator—\(c;c\). \(c^n\) can be written as \(c;c;c;\cdots;c\). The operator ';' allows the sequential composition of control operators. A sequence of operators can be delimited by { and }.

Start:
\[
\text{msg init}(v) \{ \text{select('a') } \Rightarrow v \; ; \; \text{coin(10) } \Rightarrow v \; ; \; \text{becomes Dispense} \}
\]

Assignment operator—\(x = e\). Assign expression \(e\) to local state variable \(x\).

Conditional operator—\(\text{if } (e) \; P_t \; \text{else } P_f\). The conditional operator describes an actor which behaves like \(P_t\) when the expression \(e\) evaluates to \text{true}, otherwise it behaves like \(P_f\).

\[
\text{msg coin}(y) \{ \\
\quad \text{if } (x \; = \; 'a') \; \{ \; \text{dispense('a') } \Rightarrow u \; ; \; \text{becomes Select} \; \} \\
\quad \text{if } (y \; = \; 5) \; \text{becomes VM10} \\
\quad \text{else} \; \{ \; \text{dispense('b') } \Rightarrow u \; ; \; \text{becomes Select} \; \}
\}
\]

Actor creation—\(a = \text{new } D\). Create a new actor \(a\) using actor definition \(D\) and initialize its local state \((s,s',\cdots)\) according to actor definition \(D\). All new actors begin their lifetime in conceptual state \text{Start}. The actor executing \text{new } D\ is called the owner of \(a\). The owner remains active and continues to execute its message handler.

def Main(u,v)
Start:
\[
\text{msg init}() \{ \\
\quad u = \text{new User} \; ; \; v = \text{new Vend} \; ; \\
\quad \text{init}(v) \Rightarrow u \; ; \; \text{init}(u) \Rightarrow v \; ; \; \text{terminate}
\}
\]

Send operator—\(f(e) \Rightarrow a\). Send message \(f(e)\) to actor \(a\). The message contains communication \(e\). The actor executing a send operator prepares the activation message \(\alpha = (\text{self},a,f,e)\) then continues execution of its message handler.

VM10:
\[
\text{msg coin}(y) \{ \; \text{dispense('b') } \Rightarrow u \; ; \; \text{becomes Select} \; \}
\]
**Start timer**—$\text{trigger}(f(e),n,p)$. The $\text{trigger}$ operator creates a \texttt{oneshot} or \texttt{periodic} timer (depending on the value of $p$) with a duration (period) of $n$ time units. Timers are decremented by one time unit each time $\sqrt{\text{r}}$ is accepted by a timer. When a timer $t$ expires, a timeout message $\beta = (t,a,f,e)$ is sent to actor instance $a$, the owner of the timer. On timeout, \texttt{oneshot} timers are automatically discarded and \texttt{periodic} timers are automatically retriggered.

**def** User(t)
  **Start:**
  msg init(v) {
    $t = \text{trigger}(\text{noselect()},15,\text{oneshot});$
  }

**Stop timer**—$\text{discard} \ t$. Discard timer $t$. The active actor instance discards timer $t$ then continues execution of the current message handler.

  **Dispense:**
  msg dispense(x) { discard t ; terminate }

**Current time**—now. In the following example, actor $d \in \text{Drift}$ is used by $c \in \text{Checker}$ to check the accuracy of a timer $tm$. When initialized $c$ creates and initializes $d \in \text{Drift}$ then waits for a $\text{drift()}$ response from $d$. Actor $d$ records the current time, starts a 15 unit timer and waits for the timeout. Actor $d$ accepts the timeout message and responds to $c$ with the duration of the timer.

**def** Checker(d, duration)
  **Start:**
  msg init() { $d = \text{new Drift} ; \text{init()} \Rightarrow d ; \text{commit} \}
  msg drift(x) { duration = x ; \text{commit} }
  ...
**enddef**

**def** Drift(t, tm, c)
  **Start:**
  msg init(x) { $c = * ; t = \text{now} ;$
    $\quad tm = \text{trigger}(\text{sample()},15,\text{oneshot}) ; \text{commit} \}
  msg sample() { $\text{drift}(\text{now} - t) \Rightarrow c ; \text{terminate} \}
**enddef**
Commit and change conceptual state—becomes \( q \). Commit all changes to the system state then enter conceptual state \( q \).

Start:
\[
\text{msg init}(u) \{ \text{select('a')} \Rightarrow u \text{; becomes Select} \}
\]

Abort processing—abort. Discard all changes to the system state but remain in the current conceptual state.

Dispense:
\[
\text{msg reject}(x) \{ \text{eject}(y) \Rightarrow c \text{; abort} \}
\]

In the example above, when message handler \text{reject()}\) aborts, the output message eject(y) will be discarded.

Terminate processing—terminate. Commit all changes to the system state then terminate.

Dispense:
\[
\text{msg dispense}(x) \{ \text{eject}(y) \Rightarrow c \text{; terminate} \}
\]

In the example above, when message handler \text{dispense()}\) terminates, the output message eject(y) will be committed.

2.4 Example: Designing a Vending Machine Configuration

Using our Timed Actor model a distributed application is modelled using message sequence charts [7] to capture the interactions between actors in a global configuration (see Section A.1), and communicating finite state machines to describe the behaviour of individual actors (see Section A.2). A STAL definition for a distributed application follows directly from its message sequence charts and communicating finite state machines. In this section we illustrate the design of a vending machine application.

A vending machine dispenses two items, item ‘a’ at a cost of 10 cents, and item ‘b’ at a cost of 15 cents; and accepts two denominations, a 5 cent coin, and a 10 cent coin. An item is dispensed
each time a user selects an item and enters the necessary number of coins. The time to make an item selection and enter the correct coinage cannot exceed 15 time units.

An instance of the vending machine configuration is constructed by placing actors \(u \in \text{User}\) and \(v \in \text{Vend}\) in a parallel composition, and associating timer \(t \in \text{Timer}\) with actor \(u\) (Figure 2.3). The association of timer \(t\) with its owner \(u\) is denoted \(u \upharpoonright t\).

\[
\text{System} ::= (u \upharpoonright t) \mid v \tag{2.15}
\]

![CFSM for \(u \in \text{User}\)](image1)

![CFSM for \(v \in \text{Vend}\)](image2)

Figure 2.3: A vending machine configuration.

The interaction between the vending machine components \((u \in \text{User}, v \in \text{Vend})\) is illustrated using the message sequence chart in Figure 2.4 [35]. The first message exchange in Figure 2.4 illustrates a successful use of a vending machine; a vending machine user makes an item selection and enters the correct coinage within 15 units of time. The second message exchange in Figure 2.4 illustrates an unsuccessful use of the vending machine; a vending machine user fails to enter the correct coinage within 15 units of time results, which causes a restart of the vending machine.
**Definition User—a vending machine user.** An actor instance of User is described by a *timed communicating finite state machine* $G_{\text{User}} = (Q, Q_f, q_0, T, \Sigma_{\text{User}}, B)$ (Figure 2.3(a)), where

| $Q$ | $\{\text{Start, Dispense, 0}\}$, $Q_f = \{0\}$, $q_0 = \text{Start}$, $T = \{t\}$, |
| $\Sigma_{\text{User}}$ | $\{\text{init, dispense, noselect}\}$, and |
| $B$ | is the transition relation summarized in Table 2.2. |
Table 2.2: Transition relation for actor \( u \in User \).

<table>
<thead>
<tr>
<th>( q_{\text{current}} )</th>
<th>( \Sigma_{\text{User}} )</th>
<th>( q_{\text{next}} )</th>
<th>( K^+ )</th>
<th>( J^+/J^- )</th>
<th>( X^+ )</th>
<th>( R )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>init(x)</td>
<td>Dispense</td>
<td>{ }</td>
<td>{+t}</td>
<td>{v.select(y) \cdot v.coin(5) \cdot v.coin(5)}</td>
<td>becomes Dispense</td>
</tr>
<tr>
<td>Dispense</td>
<td>dispense(x)</td>
<td>0</td>
<td>{ }</td>
<td>{−t}</td>
<td>{ }</td>
<td>terminate</td>
</tr>
<tr>
<td>*</td>
<td>noselect()</td>
<td>Start</td>
<td>{ }</td>
<td>{ }</td>
<td>{v.restart()}</td>
<td>becomes Start</td>
</tr>
</tbody>
</table>

The STAL definition for component User is given in Listing 2.1. When timer \( t \) times out, message handler noselect() sends a restart() message to \( v \in Vend \) causing \( v \) to enter state Select. In an ideal system, the triggering of this condition would force any coins in the system to be returned to the user. To keep the specification simple no coins are returned.

Listing 2.1: Actor definition User.

```python
def User(t,v):
    Start:
        msg init(x) {
            v = x;
            t = trigger(noselect(),15, oneshot);
            select('b') ⇒ v ; coin(5) ⇒ v ; coin(5) ⇒ v ; coin(5) ⇒ v ;
            becomes Dispense
        }
    Dispense:
        dispense(x) { discard t ; terminate }
    Defaults:
        noselect() { restart() ⇒ v ; becomes Start }
enddef
```

Definition Vend—a simple vending machine. An actor instance of Vend is described by communicating finite state machine \( G_{\text{Vend}} = (Q, Q_f, q_0, \Sigma_{\text{Vend}}, B) \) (Figure 2.3(b)), where

| \( Q \) = \{Start, Select, VM0, VM5, VM10\}, \( Q_f = \{\} \), \( q_0 = \text{Start} \) |
| \( \Sigma_{\text{Vend}} \) = \{init, select, coin, restart\}, and |
| \( B \) = is the transition relation summarized in Table 2.3. |

\(^2\)Column \( J^+/J^- \) is the combination of \( J^+ \) and \( J^- \), \((+t)\) indicating the creation of a timer, and \((−t)\) indicating the discard of a timer.
Table 2.3: Transition relation for actor \( v \in \text{Vend} \).

<table>
<thead>
<tr>
<th>( q_{\text{current}} )</th>
<th>( \Sigma_{\text{Vend}} )</th>
<th>Condition</th>
<th>( q_{\text{next}} )</th>
<th>( K^+ )</th>
<th>( X^+ )</th>
<th>( R )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>init()</td>
<td>Select</td>
<td>{}</td>
<td></td>
<td></td>
<td>becomes Select</td>
</tr>
<tr>
<td>Select</td>
<td>select(y)</td>
<td>VM0</td>
<td>{}</td>
<td></td>
<td></td>
<td>becomes VM0</td>
</tr>
<tr>
<td>VM0</td>
<td>coin(5)</td>
<td>VM5</td>
<td>{}</td>
<td></td>
<td></td>
<td>becomes VM5</td>
</tr>
<tr>
<td>VM0</td>
<td>coin(10) y = 'b'</td>
<td>VM10</td>
<td>{}</td>
<td></td>
<td></td>
<td>becomes VM10</td>
</tr>
<tr>
<td>VM0</td>
<td>coin(10) y = 'a'</td>
<td>Select</td>
<td>{} [u.dispense('a')]</td>
<td></td>
<td></td>
<td>becomes Select</td>
</tr>
<tr>
<td>VM5</td>
<td>coin(5)</td>
<td>VM10</td>
<td>{}</td>
<td></td>
<td></td>
<td>becomes VM10</td>
</tr>
<tr>
<td>VM5</td>
<td>coin(10) y = 'a'</td>
<td>Select</td>
<td>{} [u.dispense('a')]</td>
<td></td>
<td></td>
<td>becomes Select</td>
</tr>
<tr>
<td>VM5</td>
<td>coin(10) y = 'b'</td>
<td>Select</td>
<td>{} [u.dispense('b')]</td>
<td></td>
<td></td>
<td>becomes Select</td>
</tr>
<tr>
<td>VM10</td>
<td>coin(y)</td>
<td>Select</td>
<td>{} [u.dispense('b')]</td>
<td></td>
<td></td>
<td>becomes Select</td>
</tr>
<tr>
<td>*</td>
<td>restart()</td>
<td>Select</td>
<td>{}</td>
<td></td>
<td></td>
<td>becomes Select</td>
</tr>
</tbody>
</table>

The STAL definition for component \( \text{Vend} \) is given in Listing 2.2. Notice that this vending machine does not make change, it is possible for users to overpay for their selection.

Listing 2.2: Actor definition Vend.

```python
def Vend(x,u):
    Start:
        msg init() { becomes Select }
    Select:
        msg select(y) { u = * ; x = y ; becomes VM0 }
    VM0:
        msg coin(y) {
            if (y == 5) becomes VM5
            if (x == 'b') becomes VM10
            dispense('a') \Rightarrow u ; becomes Select
        }
    VM5:
        msg coin(y) {
            if (x == 'a') { dispense('a') \Rightarrow u ; becomes Select }
            if (y == 5) becomes VM10
            dispense('b') \Rightarrow u ; becomes Select
        }
    VM10:
        msg coin(y) { dispense('b') \Rightarrow u ; becomes Select }
    Defaults:
        msg restart() { becomes Select }
enddef
```
Transition $(VM5, \text{coin}(10), \text{Select})$ is taken when a user overpays for item ‘a’, and transition $(VM10, \text{coin}(10), \text{Select})$ is taken when a user overpays for item ‘b’. The transition $(\ast, \text{restart}(), \text{Select})$ indicates that the CFSM can transition from any state to state Select when message restart() is accepted.

**Activation history.** The initial configuration of this system consists of the actors $u$ and $v$, both in state Start, and two pending init() messages.

$$C_0 = \{ \{(u, \text{Start}), (v, \text{Start})\}, \{ \}, [u.\text{init}(v), v.\text{init}(u)] \}$$

To process the input sequence select('b')·coin(5)·coin(5)·coin(5) the vending machine evolves through a sequence of configurations. The sequence $C_1 \xrightarrow{\mu_1} C_2 \xrightarrow{\mu_2} C_3 \cdots C_{k-1} \xrightarrow{\mu_{k-1}} C_k$ is called an activation history, where $\mu \in \{\alpha, \beta, \sqrt{\cdot}\}$. A typical activation history is depicted in Table 2.4. The local state of timer $t$ is represented by its counter value.

Table 2.4: Example activation history for the vending machine configuration.

<table>
<thead>
<tr>
<th>Action ($\mu$)</th>
<th>Actors ($\chi$)</th>
<th>Timers ($\beta$)</th>
<th>Activations ($\chi$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>u.init</td>
<td>{ (u, Start), (v, Start) }</td>
<td>{ }</td>
<td>[u.\text{init}(v)·v.\text{init}(u)]</td>
</tr>
<tr>
<td>v.select</td>
<td>{ (u, Dispense), (v, Start) }</td>
<td>{ }</td>
<td>[v.\text{init}()·v.\text{select}(\text{'b'})·v.\text{coin}(5)^{(3)}]</td>
</tr>
<tr>
<td>v.coin</td>
<td>{ (u, Dispense), (v, VM0) }</td>
<td>{ t.\text{Tick}, 63 }</td>
<td>[v.\text{coin}(5)^{(3)}]</td>
</tr>
<tr>
<td>v.coin</td>
<td>{ (u, Dispense), (v, VM5) }</td>
<td>{ t.\text{Tick}, 48 }</td>
<td>[v.\text{coin}(5)^{(2)}]</td>
</tr>
<tr>
<td>v.coin</td>
<td>{ (u, Dispense), (v, VM10) }</td>
<td>{ t.\text{Tick}, 48 }</td>
<td>[v.\text{coin}(5)^{(1)}]</td>
</tr>
<tr>
<td>u.dispense</td>
<td>{ (u, 0), (v, Select) }</td>
<td>{ }</td>
<td>[u.\text{dispense}(\text{'b'})]</td>
</tr>
</tbody>
</table>
2.5 Summary

In this chapter we describe the *Timed Actor model* in the context of a configuration consisting of actors, timers, and a list of pending messages. In this model pending messages are dispatched in FIFO fashion and message processing is *atomic*.

In the *Timed Actor model* nodes are provisioned with a master-clock $T$ which generates a discrete periodic event denoted $\sqrt{}$. Timers accept $\sqrt{}$ events and react to two operations *trigger* and *discard*. *trigger* is used to create and enable a timer, *discard* stops and discards a timer. Timers allow an actor to react to the passage of time. The behaviour of an actor in a timed configuration is completely described by the set of actors it creates, the set of timers it manages, the sequence of output messages it generates, and its choice of a commitment operator.

The *Timed Actor model* differs from the *Actor model* [1] in other important ways.

- The nature of replacement behaviour. In the *Actor model* replacement behaviour is defined by the *become* operator; in our *Timed Actor model* replacement behaviour is expressed by the commitment operators (*becomes*, *terminate*, and *abort*) which implement a local checkpointing mechanism.

- The use of conceptual states. In our *Timed Actor model* the *becomes* operation allows an actor to transition between named conceptual states. This has the effect of enabling and disabling specific messages. The *Actor model* has no such facility.

An informal semantics of the *Timed Actor model* is presented in the form of a *simple timed actor language* (STAL). This informal semantics identifies the programming language constructs necessary to define a *timed actor language*. We illustrate how message sequence charts and communicating finite state machines describing actor behaviour can be used to specify STAL actor definitions for a vending machine system.
Chapter 3

An Asynchronous Actor Algebra

This chapter describes ACube, an asynchronous actor algebra. The design of ACube is influenced by the algebra of actors described by Gaspari and Zavattaro [19], and the Actor models of Agha [1] and Frølund [5]. The asynchronous π-calculus of Honda and Tokoro [24, 36, 37, 38] provides a foundation for understanding asynchronous algebras.

ACube provides a model of concurrency which differs from the treatments in [1, 24, 36, 20, 5, 19] in the following ways:

1. ACube supports three categories of first-class objects; actors, timers, and messages. Timers allow an actor to react to the passage of time.

2. ACube supports a class definition facility; the message interface of a class definition is structured as a communicating finite state machine.

3. ACube message processing is atomic, it either succeeds or fails. This behaviour is facilitated by commitment operators that implement a local checkpointing mechanism.

4. Every actor and timer instance in ACube possesses a globally unique identity, and a private local state.

When we compare the treatments of the process algebras and calculi in [39, 40, 24, 36, 41] to the treatments of Actor models in [1, 5, 19] we find that Actor models provide a number of
important specializations. The models presented by [1, 5, 19] provide an operator for determining the replacement behaviour of an executing actor, while the models of [5, 19] describe actors with a private local state. The timed actor semantics introduced in this chapter provides additional specialization; timers are introduced as first-class objects allowing actors to react to the passage of time, message processing is atomic and, actors behave like communicating finite state machines with input messages causing state transitions. These specializations result in a model which is more expressive than those presented by [39, 40, 24, 36, 41].

Outline. The goal of this chapter is to develop an operational semantics for a programming language that adheres to our Timed Actor model introduced in Section 2.2. We refer to this semantics as a timed actor semantics. Our treatment focuses on the semantics of commitment (Section 3.4), asynchronous communication (Section 3.5), actor creation (Section 3.6), and timers (Section 3.7). We develop a timed actor semantics by examining the transitional semantics and algebraic properties of ACube. The transitional semantics provides the meaning of the basic operators, and a mechanism for describing a labelled transition system. Following [39] we examine the algebraic properties of ACube by describing some of the algebraic laws which apply to the basic ACube operators. ACube operators fall into two principal categories; static operators describe the overall structure of ACube expressions while dynamic operators describe how interactions between actors and timers can be modelled.

3.1 A Grammar for ACube Expressions

ACube expressions are organized as a sequence of message blocks, each message block defining the messages an actor may accept in a specific conceptual state. ACube messages are named and communications may contain multiple parameters. For example, in the definition of class User in Figure 3.1, there are two conceptual states labelled Start and Dispense. The definition of conceptual state Start contains a definition for the private local state of an actor \( u \in \text{User} \), denoted \((t,v)\); and a definition for message handler init(x). The message block associated with conceptual state
**Dispense** has definitions for message handlers `dispense_a()` and `dispense_b()`. Message block **Defaults** defines behaviour that is common to all conceptual states, but **Defaults** itself is not a conceptual state.

| `Start(t,v)` | `def` | `rcv(*,init(x)) · \{v ← x\} · trigger(t,noselect(),15,oneshot)` · `send(v,select_b()) · send(v,coin_5()) · send(v,coin_5()) · becomes(Dispense)` |
| `Dispense` | `def` | `rcv(v,dispense_a()) · skip · discard(t) · terminate` + `rcv(v,dispense_b()) · skip · discard(t) · terminate` |
| `Defaults` | `def` | `rcv(t,noselect()) · send(v,restart()) · becomes(Start)` |

Figure 3.1: *ACube* actor expression for class User.

Each message handler begins with a receive prefix of the form `rcv(b,f(v))` and ends with a commitment operator. For example, in conceptual state **Dispense** actor $u \in User$ will only accept message `dispense_a()` from actor $v$, denoted `rcv(v,dispense_a())`; or message `dispense_b()` from actor $v$, denoted `rcv(v,dispense_b())`. Accepting message `dispense_a()` results in the execution of the body of message handler `dispense_a()` which completes with the execution of commitment operator `terminate`. The *ACube* commitment operator `becomes` allow an actor to change conceptual states.

In Section 3.1.1 we describe the syntactic domains used in the *ACube* grammar, in Section 3.1.2 we describe actor expressions and introduce the static operators, and in Section 3.1.3 we describe actor class definitions and introduce the dynamic operators.

### 3.1.1 Syntactic Domains

Let $A$, with typical elements $a, a', b, b', \cdots$ be a finite set of actor identities; $T$, with typical elements $t, t', \cdots$ be a finite set of timer identities; $Q$, with typical elements $q, q_0, q_1, \cdots$ be a finite set of conceptual state names; $\Sigma$, with typical elements $f, g, h, \cdots$ be a finite set of message names; $\mathbb{N}$, with
typical elements $n,n',n'',\cdots$ denote natural numbers; and $p \in \{\text{oneshot}, \text{periodic}\}$ denote timer types. We use $\sigma \in \Sigma$ to denote an unspecified message. The constant $\text{none}$ denotes a nonexistent actor or timer identity.

Let $V$, with typical elements $v,v',v'',\cdots$ denote message contents; and $S$, with typical elements $s,s',s'',\cdots$ denote the local state of an actor. We denote instance variables by $x,y,z,\cdots \in (V \cup S)$; and require the names of state variables and message parameters to be distinct, and defined without type information. State variables may be assigned actor identities, timer identities, and natural numbers. Message parameters may only be assigned actor identities; most importantly, timer identities cannot be assigned to message parameters since the ownership of a timer can never be transferred (see Section 3.3.2).

We assume that expressions $e,e',e'',\cdots$ can be constructed from natural numbers, actor identities, timer identities, timer types, the names of conceptual states, message names, the names of local state variables, the names of message parameters, and any operator we may require. $[e]$ denotes the evaluation of expression $e$.

### 3.1.2 Actor Expressions

At any instant an actor can either be ready or active [20, 19]. A ready actor is one which is able to receive its next message. An active actor is one which is currently executing a message handler. In $ACube$, actors have a globally unique identity and possess a private local state, so it is natural to denote a ready actor by instance term $\alpha_q^a(D)_s$ and an active actor by instance term $\alpha_q^a[D]_{s}^{\sigma v}$ [20], where $a$ is an actor identity, $q$ is the current conceptual state of $a$, $s$ is the current local state of $a$, $\sigma v$ is the current input message, and $D$ is an actor definition, which specifies the behaviour of an actor. $\alpha_q^a(D)_s$ is an alternative representation of an actor instance with identity $a$, as defined in Equation 2.4 on page 28.

$$K_a = (a,q,s) = \alpha_q^a(D)_s \quad (2.4')$$

In $ACube$, timers have a globally unique identity, possess a private local state, and must be associated with an actor. We denote a ready timer by instance term $\beta_n^t_a(D)_{n,p}$ and an active timer by
instance term \( t_a[D]_{n,p} \), where \( t \) is a timer identity, \( a \) is the identity of the owner of \( t \), \( \beta \) is the timeout message to be sent to the owner of \( t \) when the timer expires, \( n \) denotes the counter of \( t \), \( p \) denotes the type of \( t \), and \( D \) specifies timer behaviour. \( t_a[D]_{n,p} \) is an alternative representation of a timer instance with identity \( t \), as defined in Equation 2.7 on page 30.

\[
J_t = (t,q,(\beta,n,p)) = t_a[D]_{n,p} \quad (2.7')
\]

Let \( E \), with typical terms \( A, A', \ldots; B, B', \ldots; T, T', \ldots; M, M', \ldots \), be a finite set of actor expressions defined recursively using the following abstract grammar:

\[
A ::= a_q(D) | \sum a_q[D]^\sigma_v \mid A|A \mid A \upharpoonright T \mid A|M \mid 0 \\
T ::= t_a[D]_{n,p} | \sum t_a[D]_{n,p} \mid T|T \mid 0 \\
M ::= \langle b a \sigma v \rangle \mid \langle t a \sigma v \rangle \mid \langle 0 \rangle
\] (3.1)

Terms \( A, A', B, B' \) are referred to as actor terms; terms \( T, T' \) are referred to as timer terms; and terms \( M, M' \) are referred to as message terms. The instance terms \( a_q(D) \), \( a_q[D]^\sigma_v \), \( t_a[D]_{n,p} \), \( t_a[D]_{n,p} \), \( \langle b a \sigma v \rangle \), \( \langle t a \sigma v \rangle \), \( 0 \), and \( \langle 0 \rangle \) are the terminals of the grammar in Equation 3.1.

ACube supports two static operators, composition; \( A|A, T|T \), and \( A|M \) (Section 3.3.1), and association; \( A \upharpoonright T \) (Section 3.3.2). ACube messages are objects denoted by \( \langle b a \sigma v \rangle \) or \( \langle t a \sigma v \rangle \); where \( b \) is the identity of the source actor, \( t \) is the identity of the source timer, \( a \) is the identity of a destination actor, \( \sigma \) is a message name, and \( v \) is a communication (message contents). We denote a null actor or null timer by \( 0 \); and a null message by \( \langle 0 \rangle \).

### 3.1.3 Actor Class Definitions

Let \( D_C \) be a family of actor definitions which range over an index set \( C \); then \( D_1, \ldots, D_C, \ldots, D_n \) are referred to as actor classes. Every actor class \( D_C \) is provided with a behavioural definition of the form

\[
D_C(S_C) \overset{\text{def}}{=} \sum \sum \text{rcv}(b, \sigma_j(v)) \cdot P_{ij}
\] (3.2)

where \( S_C \) is a local state definition for class \( D_C \), \( \text{rcv}(b, \sigma_j(v)) \) is a receive prefix for message \( \sigma_j \in \Sigma_C \) in conceptual state \( i \in Q_C \), `\cdot` is a sequence operator, and \( P_{ij} \) is the definition of a message handler for
message $\sigma_{ij}$. Actor class definitions allow a message $\sigma(v)$ to be redefined in any conceptual state. In a receive operation, $b$ is referred to as an “input guard”. A message $\langle b' a \sigma v \rangle$ is accepted by a destination actor $a$ when $b = b'$, where $b'$ is the identity of the actor (timer) originating the message. When message $\sigma_{ij}(v)$ can be accepted from any actor (timer) we write the receive operation as $\text{rcv}(\star, \sigma_{ij}(v))$.

A message handler $P$ is constructed from two categories of dynamic operators. Control operators ($c$) execute immediately leaving an active actor in the active state. Commitment operators ($r$) either terminate the active actor or return it to the ready state. Execution of message handlers is atomic, once an actor accepts a message it will not be preempted until it reaches a commitment point (executes a commitment operator $r$). Message handlers are constructed using the following grammar, where $c^*$ denotes 0 or more control operations and $r$ denotes a commitment operator.

$$
P \ ::= \ c^* \cdot r
$$

$$
P_t, P_f \ ::= \ P \mid c^*
$$

$$
c \ ::= \ \text{skip} \mid \{ x \leftarrow e \} \mid P_t \ \text{\Leftrightarrow} \ e \ \text{\Leftrightarrow} \ P_f \mid \text{new}(b, C)
$$

$$
(3.3)
$$

$$
r \ ::= \ \text{becomes}(q) \mid \text{abort} \mid \text{terminate}
$$

A message handler $P$ always takes the form $c^* \cdot r$, where $c^*$ can be written as $c \cdot c \cdot c \cdot \cdots \cdot c$. In the conditional operator, $P_t \ \text{\Leftrightarrow} \ e \ \text{\Leftrightarrow} \ P_f$, blocks $P_t$ and $P_f$ may take either the $c^* \cdot r$ or $c^*$ form. ACube supports the following dynamic operators: $\text{new}(b, C)$ creates a new actor instance $b$ with class behaviour $D_C$, $\text{trigger}(t, f(e), n, p)$ creates and enables a timer $t$ of type $p$ (oneshot or periodic) with duration $n$ and timeout message $f(e)$, $\text{discard}(t)$ destroys timer $t$, $\text{send}(b, f(e))$ sends message $f(e)$ to actor $b$, $\text{forward}(b)$ forwards the current activation message to actor $b$, $\text{becomes}(q)$ finalizes message processing allowing an active actor to enter conceptual state $q$, $\text{terminate}$ finalizes message processing allowing an active actor to self-terminate, $\text{abort}$ aborts message processing allowing the rollback of a configuration to the point where the current input message was accepted, and $\{ x \leftarrow e \}$ assigns the value of expression $e$ to local state variable (or message parameter) $x$. $\text{skip}$ denotes the null operation. $\text{skip}$ has no observable effect on the executing actor and is often used as
a placeholder for unspecified behaviour.

### 3.2 A Transition System for Actor Expressions

Timed actor configurations provide a suitable abstraction for describing the coarse-grained changes in actor systems originating from specified actor behaviour (see pages 32–36). The transitional semantics for ACube provides a suitable abstraction for describing the fine-grained changes in actor systems; namely, they allow us to model how actor behaviour can be realized.

In Sections 3.3–3.8 we describe the meaning of each ACube static and dynamic operator. In each section we introduce the transition and algebraic rules which apply to each operator. A structured operational semantics for ACube expressions (\( E \)) follows directly from the definition of each transition rule. An operational semantics for ACube is summarized in Table 3.2 on page 56 and Table 3.3 on page 66.

**Definition 3.1 (ACube Action Set)** The action set accepted by actor and timer terms in an ACube expression \( E \) is given by

\[
Act = \{\tau, \sqrt{\text{ }}\} \cup \{ba\sigma v, \overline{ba\sigma v} \mid a,b \in A, \sigma \in \Sigma, v \in V\} \tag{3.4}
\]

where \( \{\tau, \sqrt{\text{ }}\} \) are internal actions; \( \tau \) is the silent action, and \( \sqrt{\text{ }} \) is the clock tick generated by a master-clock \( T \). The pattern \( \overline{ba\sigma v} \) denotes the transmission of message \( \sigma(v) \) by actor \( b \) to recipient \( a \), and \( ba\sigma v \) denotes the reception of message \( \sigma(v) \) by recipient \( a \) from actor \( b \). The actions \( \alpha, \beta, \cdots \) range over \( Act \).

Labelled transition systems describe the evolution of actor expressions.

**Definition 3.2 (Labelled Transition System)** The labelled transition system modelling timed actor behaviour is defined by the triple \( (E, Act, \xrightarrow{\alpha}) \), where \( E \) is a set of ACube expressions (states), \( Act \) is the set of labels, and \( \xrightarrow{\alpha} \subseteq E \times Act \times E \) is the transition relation satisfying the transition rules in Table 3.2 on page 56 and Table 3.3 on page 66, and the algebraic properties described in Proposition 3.1–Proposition 3.4.
3.3 Static Operators

Static operators impose structure in ACube expressions. In ACube static operators are represented by composition and association.

3.3.1 Composition

\( A | A \) denotes the parallel composition of actors. Two actors in a parallel composition either evolve independently (Par) or interact when one actor accepts a message originating from a second actor (ASync). The sending of a message does not block the originating actor and reception of a message by the destination actor has no effect on the sender of a message.

\[
\begin{align*}
\text{[Par]} & : \\
A & \xrightarrow{\alpha} A' \\
A & \xrightarrow{\alpha} A'|B \\
B & \xrightarrow{\alpha} B' \\
A & \xrightarrow{\alpha} A'|B' \\
\end{align*}
\]

\[
\begin{align*}
\text{[ASync]} & : \\
A & \xrightarrow{ab\sigma v} A' \\
A & \xrightarrow{ab\sigma v} A'|\langle ab\sigma v \rangle |B \\
B & \xrightarrow{ab\sigma v} B' \\
B & \xrightarrow{ab\sigma v} B'|A' \langle \{0\} \rangle |B' \\
\end{align*}
\]

\( T | T \) denotes the parallel composition of timers. All timers in a local configuration either evolve independently (TPar) or synchronize on the special action \( \sqrt{} \) (TSync).

\[
\begin{align*}
\text{[TPar]} & : \\
T_1 & \xrightarrow{\alpha} T'_1 \\
(T_1 | T_2) & \xrightarrow{\alpha} (T'_1 | T_2) \\
T_2 & \xrightarrow{\alpha} T'_2 \\
(T_1 | T_2) & \xrightarrow{\alpha} (T'_1 | T_2) \\
\text{if } \alpha & \neq \sqrt{} \\
\end{align*}
\]

\[
\begin{align*}
\text{[TSync]} & : \\
T_1 & \xrightarrow{\sqrt{}} T'_1 \\
T_1 | T_2 & \xrightarrow{\sqrt{}} T'_1 | T_2 \\
\cdots & \\
T_n & \xrightarrow{\sqrt{}} T'_n \\
(T_1 | T_2 | \cdots | T_n) & \xrightarrow{\tau} (T'_1 | T'_2 | \cdots | T'_n) \\
\end{align*}
\]

The term \( A | M \) denotes the parallel composition of an actor term with a message term; a situation which arises in transition rules ASync, ATSync, Accept, Reject, OneShotTimeout, and PeriodicTimeout.

**Proposition 3.1 (Composition Laws)**

1. Actor terms freely commute and associate over composition. The null actor \( \emptyset \) acts as an identity in actor expressions involving composition.

\[
A | \emptyset = A, \quad A | B = B | A, \quad \text{and} \quad (A_1 | A_2) | B = A_1 | (A_2 | B)
\]
2. Timer terms freely commute and associate over composition. The null timer $\mathbf{0}$ acts as an identity in actor expressions involving composition.

$$
T | \mathbf{0} = T, \quad T_1 | T_2 = T_2 | T_1, \quad \text{and} \quad (T_1 | T_2) | T_3 = T_1 | (T_2 | T_3)
$$

3. Message terms commute with actor terms. The nature of commitment in ACube prohibits message terms of the form $\mathcal{M}_1 \cdots | \mathcal{M}_n$ for $n > 1$ (see Section 3.4). The null message $\langle \mathbf{0} \rangle$ acts as an identity in actor expressions involving composition.

$$
\mathcal{A} | \langle \mathbf{0} \rangle = \mathcal{A}, \quad \mathcal{A} | \mathcal{M} = \mathcal{M} | \mathcal{A}
$$

3.3.2 Association

$(\mathcal{A} | \mathcal{T})$ is the parallel composition of ready and active actors with ready and active timers. In the term $(\mathcal{A} | \mathcal{T})$, $\mathcal{A}$ is referred to as the owner of timer $\mathcal{T}$. An associated actor and timer either evolve independently (ATPar) or they interact when an actor accepts a timeout message from one of its timers (ATSync). The sending of a timeout message does not block the originating timer and reception of a timeout message by the owner has no effect on the associated timer.

$$
\text{ATPar:} \quad \frac{\mathcal{A} \xrightarrow{\alpha} \mathcal{A}'}{(\mathcal{A} | \mathcal{T}) \xrightarrow{\alpha} (\mathcal{A}' | \mathcal{T})} \quad \frac{\mathcal{T} \xrightarrow{\alpha} \mathcal{T}'}{(\mathcal{A} | \mathcal{T}) \xrightarrow{\alpha} (\mathcal{A} | \mathcal{T}')}
$$

$$
\text{ATSync:} \quad \frac{\mathcal{T} \xrightarrow{t a \sigma v} \mathcal{T}'}{(\mathcal{A} | \mathcal{T}) \xrightarrow{t a \sigma v} (\mathcal{A} | \mathcal{T}') | \langle t a \sigma v \rangle} \quad \frac{\mathcal{A} \xrightarrow{t a \sigma v} \mathcal{A}'}{(\mathcal{A} | \mathcal{T}) | \langle t a \sigma v \rangle \xrightarrow{t a \sigma v} (\mathcal{A}' | \mathcal{T}) | \langle \mathbf{0} \rangle}
$$

Proposition 3.2 (Timer Association Laws)

1. Timers cannot function in the absence of their owners. Every implementation of a timed actor semantics is responsible for discarding timers no longer associated with an actor.

$$
(\mathcal{A} | \mathbf{0}) = \mathcal{A}, \quad \text{and} \quad (\mathbf{0} | \mathcal{T}) = \mathbf{0}
$$

2. Members of a timer association do not commute; actors own timers, timers do not create actors. Actors cannot commute across a timer association; the ownership of a timer is never
transferred.

\[ (A \upharpoonright \tau) \neq (\tau \upharpoonright A) \]
\[ A \upharpoonright (B \upharpoonright \tau) \neq B \upharpoonright (A \upharpoonright \tau) \]

3. Timer associations commute with message terms.

\[ (A \upharpoonright \tau) \upharpoonright M = M \upharpoonright (A \upharpoonright \tau) \]

4. Since every timer has an associated actor, terms of the form \( \tau \upharpoonright M \) denoting the composition of timer and message terms are not supported. We assert that

\[ (A \upharpoonright \tau) \upharpoonright M \neq (A \upharpoonright \tau) \upharpoonright (M \upharpoonright \tau) \]

since only actors, not messages, may own timers; and ownership does not apply to messages.

\[
\begin{align*}
\text{[Par]} & : & A \xrightarrow{\alpha} A' & \quad B \xrightarrow{\alpha} B' & \quad \alpha \neq \sqrt{} & \quad A\upharpoonright B \xrightarrow{\alpha} A'\upharpoonright B' \\
\text{[ASync]} & : & \overline{A \parallel b\sigma_v} & \quad \overline{A'} \quad \overline{B \parallel b\sigma_v} & \quad \overline{B'} & \quad A \upharpoonright (b\sigma_v) \parallel B \xrightarrow{\alpha} A' \parallel (b\sigma_v) \parallel B' \\
\text{[TPar]} & : & \overline{T_1 \alpha} & \quad \overline{T'_1} \quad \overline{T_2 \alpha} & \quad \overline{T'_2} & \quad \overline{\cdots} & \quad \overline{T_n \alpha} & \quad \overline{T'_n} & \quad \text{if } \alpha \neq \sqrt{} & \quad \overline{(T_1 \parallel T_2 \parallel \cdots \parallel T_n)} \xrightarrow{T} \overline{(T'_1 \parallel T'_2 \parallel \cdots \parallel T'_n)} \\
\text{[TSync]} & : & \overline{T_1 \tau} & \quad \overline{T'_1} \quad \overline{T_2 \tau} & \quad \overline{T'_2} & \quad \overline{\cdots} & \quad \overline{T_n \tau} & \quad \overline{T'_n} & \quad \text{if } \alpha \neq \sqrt{} & \quad \overline{\tau} \xrightarrow{T} \overline{T'} \\
\text{[ATPar]} & : & A \xrightarrow{\alpha} A' & \quad \overline{T \alpha} & \quad \overline{T'} & \quad \overline{T} \xrightarrow{\alpha} \overline{T'} & \quad (A \upharpoonright \tau) \xrightarrow{\alpha} (A' \upharpoonright \tau) \\
\text{[ATSync]} & : & \overline{\tau} \xrightarrow{\alpha} \overline{T} & \quad \overline{T'} & \quad \overline{(a \sigma_v)} & \quad \overline{(A \upharpoonright \tau) \parallel (a \sigma_v)} & \quad \overline{\tau} \xrightarrow{\alpha} \overline{T'} & \quad (A \upharpoonright \tau) \parallel (a \sigma_v) \xrightarrow{\alpha} (A' \upharpoonright \tau) \parallel (a \sigma_v) \\
\end{align*}
\]

Figure 3.2: An operational semantics for ACube static operators.
3.4 Commitment

To support atomic message processing in the timed actor semantics we utilize an intentions list \( \mathcal{I} \) to record pending changes to a local configuration \( C \) resulting from the activation of an actor \( a \). The context of each actor activation, called a checkpoint \((\omega, q, s_c, \mathcal{I})\), consists of the activation time \( \omega \) of actor \( a \), the conceptual state \( q \) of actor \( a \) at activation time, a copy of the local state \( s_c \) of actor \( a \) at activation time, and the intentions list \( \mathcal{I} \). Only the intentions list \( \mathcal{I} \), initially empty, is modifiable during message processing.

The intentions list is modified by the \texttt{new}, \texttt{send}, \texttt{forward}, \texttt{trigger} and \texttt{discard} operators. In transition rules for control operators, appending an object instance \( \kappa \) to the intentions list \( \mathcal{I} \) is denoted \( \mathcal{I} \leftarrow \kappa \), where \( \kappa \) is one of the object instances listed in Table 3.1. A transition rule for a control operator \( c \) takes the form

\[
[\text{Control}] : (\omega, q, s_c, \mathcal{I}) \vdash a[c \cdot P] \xrightarrow{\tau} (\omega, q, s_c, \mathcal{I} \leftarrow \kappa) \vdash a[P]_{s'}
\]

Rule Control reads “if control operation \( c \) completes, object instance \( \kappa \) will be appended to the intentions list \( \mathcal{I} \) then actor \( a \) will continue execution of message handler \( P \). The execution of control operation \( c \) may modify the local state of active actor \( a \)”. A change to the local state of an actor is denoted by \( [\cdots]_s \xrightarrow{\tau} [\cdots]_{s'} \).

Table 3.1: Object instances stored on the intentions list.

<table>
<thead>
<tr>
<th>Instance (( \kappa ))</th>
<th>Partition of ( \mathcal{I} )</th>
<th>Instance Type</th>
<th>Description</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathcal{I} ) ( \mathcal{I} )</td>
<td>( K^+ )</td>
<td>actor instance</td>
<td>actor creation</td>
<td>New on page 62</td>
</tr>
<tr>
<td>( \mathcal{I} ) ( \mathcal{I} ) ( \mathcal{I} )</td>
<td>( J^+ )</td>
<td>timer instance</td>
<td>timer creation</td>
<td>Trigger on page 62</td>
</tr>
<tr>
<td>( \mathcal{I} ) ( \mathcal{I} ) ( \mathcal{I} )</td>
<td>( J^- )</td>
<td>timer instance</td>
<td>timer destruction</td>
<td>Discard on page 62</td>
</tr>
<tr>
<td>( \langle ab f v \rangle )</td>
<td>( X^+ )</td>
<td>message instance</td>
<td>message generation</td>
<td>Send on page 61</td>
</tr>
<tr>
<td>( \langle ab \sigma v \rangle )</td>
<td>( X^+ )</td>
<td>message instance</td>
<td>message forwarding</td>
<td>Forward on page 61</td>
</tr>
</tbody>
</table>

The processing of an input message \( \alpha \) by an actor \( a \) results in a transition \( C \xrightarrow{\alpha} C' \) in a
local actor configuration, where $C'$ is configuration $C$ with input message $\alpha$ removed from the list of pending messages. In transition $C \xrightarrow{\alpha} C' \uplus \mathcal{J}$ operator $\uplus$ combines configuration $C'$ with the contents of intentions list $\mathcal{J}$. The configuration update operator $\uplus$ is described on page 34 (Equations 2.11–2.13).

If message processing is successful, the intentions list $\mathcal{J}$ is combined with the local configuration $C'$, and actor instance $a$ is updated. Commitment operator $\text{becomes}(q')$ causes an actor to transition to conceptual state $q'$ and become ready (Becomes). Actor instances self-terminate using commitment operator $\text{terminate}$ (Terminate).

$$C \xrightarrow{\alpha} C' \uplus \mathcal{J}, \text{ where } K_a = a_{q'}(D)_{s'}$$

(3.11)

$$C \xrightarrow{\alpha} C' \uplus \mathcal{J}, \text{ where } K_a = a(0)$$

(3.12)

If message processing is unsuccessful, the local configuration is set to $C'$, and actor instance $a$ is restored from its checkpoint. Commitment operator $\text{abort}$ causes an actor to become ready (Abort).

$$C \xrightarrow{\alpha} C', \text{ where } K_a = a_{q'}(D)_{s'}$$

(3.13)

**Proposition 3.3 (Commitment Laws)**

1. Commitment operators are always the last operation executed in a message handler; any trailing operations are ignored.

$$r \cdot P \equiv r, \text{ } r \in \{\text{becomes, terminate, abort}\}$$

2. The commitment operator $\text{becomes}(\text{current})$ can be replaced by operator commit.

$$\text{becomes}(\text{current}) \equiv \text{commit}$$
3.5 Asynchronous Communications

3.5.1 Transition Rules Accept and Reject

Actors move between the ready and active state as messages are received. A message \( \langle b' a \sigma_j v \rangle \) is accepted by target actor \( a \) if it satisfies the acceptance conditions in Definition 3.3.

\[
\text{[Accept]} : \frac{a}{q} \left( \sum_{i \in \Sigma_{a,q}} \text{rcv}(b, \sigma_i(v)) \cdot P_i(x) \right) \quad | \quad \langle b' a \sigma_j v \rangle \xrightarrow{b' a \sigma_j v} \quad (3.14)
\]

The intentions list \( \mathcal{I} \) is reset and the local state of actor \( a \) is checkpointed (denoted \( (\omega, q, s_c, []) \) ) each time a message is accepted. Once actor \( a \) accepts message \( \sigma_j \) the input parameters are unmarshalled from the message contents \( \theta_a = \{x_1 \leftarrow v_1, x_2 \leftarrow v_2, \cdots, x_n \leftarrow v_n\} \), and the corresponding message handler \( P_j(x) \) is invoked. Actor \( a \) continues to execute \( P_j \) until it reaches a commitment point.

A message \( \langle b' a \sigma_j v \rangle \) is rejected by a target actor \( a \) if it does not satisfy the acceptance conditions in Definition 3.3.

\[
\text{[Reject]} : \frac{a}{q} \left( \sum_{i \in \Sigma_{a,q}} \text{rcv}(b, \sigma_i(v)) \cdot P_i(x) \right) \quad | \quad \langle b' a \sigma_j v \rangle \xrightarrow{b' a \sigma_j v} \quad \frac{a}{q} (D) \quad | \quad (0) \quad (3.15)
\]

Definition 3.3 (Acceptance Condition) A message \( \langle b' a \sigma_j v \rangle \) is accepted by target actor \( a \) in conceptual state \( q \) when

- \( C_1 : \sigma_j \in \Sigma_{a,q} \) — message \( \sigma_j \) is in the input set of actor \( a \) when actor \( a \) is in conceptual state \( q \), and
- \( C_2 : b' = b \) — the source \( b' \) of the message matches the input guard \( b \).

3.5.2 The Nature of Actor and Timer Communication

Message passing is the only mechanism of interaction between actors. In the ACube environment each actor possesses a globally unique identity and actors interact by exchanging messages, utilizing actor identities as addresses. The environment establishes a unicast channel between any two communicating actor instances.
Message transmission results in an implicit parallel composition of the originating actor with the generated message.

\[
\text{send}(b, \sigma(v)) \cdot A \xrightarrow{ab\sigma v} A
\]

In Equation 3.16 actor \(A\) (with identity \(a\)) sends a message \(\sigma(v)\) to actor \(B\) (with identity \(b\)). Once the message is transmitted actor \(A\) continues execution independent of \(\langle ab \sigma v \rangle\) and actor \(B\).

At some time in the future, message \(\langle ab \sigma v \rangle\) will be delivered to actor \(A\). Actor \(A\) takes no part in the delivery. This type of interaction is called asynchronous communication.

\[
\text{rcv}(a, \sigma(v)) \cdot B \xrightarrow{ab\sigma v} B
\]

The interaction between an actor and a timer is very similar to the interaction between actors. Timers are associated with the actor that creates them, and an actor may create as many timers as it requires. There is no interaction between individual timers. Each timer possesses a globally unique identity. As with actors, timer identities are utilized as addresses. When timers expire they interact with their owners by sending timeout messages. The environment establishes a unicast channel between an actor and its timer.

When a timeout occurs, timer \(T\) (with identity \(t\)) sends a timeout message \(\langle ta \sigma v \rangle\) to its owner \(A\) (with identity \(a\)). Once the message is transmitted timer \(T\) continues execution independent of \(\langle ta \sigma v \rangle\) and actor \(A\).

\[
\text{send}(a, \sigma(v)) \cdot T \xrightarrow{ta\sigma v} T
\]

At some time in the future, message \(\langle ta \sigma v \rangle\) will be delivered to actor \(A\). Timer \(T\) takes no part in the delivery. This type of interaction is also called asynchronous communication.

\[
\text{rcv}(t, \sigma(v)) \cdot A \xrightarrow{ta\sigma v} A
\]

\subsection{Message Transmission Operations}

Operation \textbf{send}(\(b, f(e)\)) transmits message \(f\) with message contents \(e\) to actor \(b\). When the \textbf{send} operation is 	extit{enabled} (see Definition 3.4), actor \(a\) prepares output message \(\langle ab f v \rangle\) which includes...
the *marshalling*, \( \theta_m = \{ v_1 \leftarrow [e_1], v_2 \leftarrow [e_2], \ldots, v_n \leftarrow [e_n] \} \), of the evaluations of message arguments ([\( e_1 \]), [\( e_2 \]), \ldots, \([e_n] \)), to the message contents \((v_1, v_2, \ldots, v_n)\). Actor \( a \) then continues execution of message handler \( P \) (Send).

\[
[\text{Send}] : (\omega, q, s_c, \mathcal{I}) \vdash^a [\text{send}(b, f(e)) \cdot P]_s \xrightarrow{\tau} (\omega, q, s_c, \mathcal{I} \cup \langle abf\rangle) \vdash^a [P]_s \quad (3.20)
\]

Operation *forward*(\( b \)) forwards the activating message of actor \( a \) to actor \( b \). When the *forward* operation is *enabled* (see Definition 3.4), actor \( a \) prepares the output message \( \langle *b\sigma v \rangle \), then continues execution of message handler \( P \) (Forward).

\[
[\text{Forward}] : (\omega, q, s_c, \mathcal{I}) \vdash^a [\text{forward}(b) \cdot P]_{s}^{\sigma v} \xrightarrow{\tau} (\omega, q, s_c, \mathcal{I} \cup \langle *b\sigma v \rangle) \vdash^a [P]^{\sigma v}_s \quad (3.21)
\]

The identity of the actor originating the message \( * \) is not modified by the *forward* operation, so a receiving actor \( b \) has no knowledge of the role played by actor \( a \). Actor \( a \) is free to modify the parameters of the activation message before a *forward* operation is executed (see Equation 3.31 on page 65).

**Definition 3.4 (Transmission Condition)** A *send* \((f = \sigma)\) or *forward* operation is enabled for transmission when

- \( C_1 : b \neq \text{none} \) — \( b \) is a valid actor identity, and
- \( C_2 : \sigma \in \Sigma_b \) — message \( \sigma \) is in the input set of actor \( b \).

Ultimately, the “legality” of a transmission is determined by certain compile-time checks, and occasionally by a combination of compile-time and run-time checks which depend on the identity of destination actor \( b \) (see Definition 5.2 on page 112).

### 3.6 Actor Creation

Operation \( \text{new}(b, \mathcal{C}) \) creates a new actor instance, assigning a *fresh\(^1\) actor identity to instance variable \( b \) in the local state of \( a \), and initializing the local state of the new actor to \( s_0 \). The behaviour of

\(^1\)A *newly created* actor or timer identity is referred to as *fresh*; \( b \not\in \mathcal{X} \) and \( t \not\in \mathcal{J} \).
actor $b$ is taken from actor class definition $D_C$. New actors begin their lifetime in conceptual state $q_0$. The \textbf{new} operation initializes protected instance variables \{\texttt{self $\leftarrow b$}\} and \{\texttt{owner $\leftarrow a$}\} in the local state of actor $b$, then actor $a$ continues execution of message handler $P$ (New). At the time of creation actor $b$ is known only to its owner, actor $a$.

$$[\text{New}] : (\omega, q, s_c, j) \vdash a \left[ \texttt{new(b, C)} \cdot P \right]_s \xrightarrow{\tau} (\omega, q, s_c, j \overset{b}{\sim} (D_C)_{q_0}) \vdash a \left[ P \right]_{s'} \quad (3.22)$$

### 3.7 Semantics of Timers

Timers are created and initialized in a single atomic operation. Operation \texttt{trigger($t, f(e), n, p$)} creates or retriggers a \textbf{oneshot} timer with a duration of $n$ time units, or a \textbf{periodic} timer with a period of $n$ time units. The owner of timer $t$ registers message $f(v)$ to handle timeout events, where $f$ is the name of the timeout message, and the \textit{marshalling}, $\theta_m = \{v_1 \leftarrow [e_1], v_2 \leftarrow [e_2], \ldots, v_n \leftarrow [e_n]\}$, prepares the future communication. The \texttt{trigger} operation has no effect when $n$ is set to 0, or when $f$ is not in the input set of actor $a$ ($f \not\in \Sigma_a$).

The \texttt{trigger} operation initializes instance variable $t$ in the local state of actor $a$ with a \textit{fresh} timer identity, enables timer $t$, then actor $a$ continues execution of message handler $P$ (Trigger).

$$[\text{Trigger}] : (\omega, q, s_c, j) \vdash a \left[ \texttt{trigger(t, f(e), n, p)} \cdot P \right]_s \xrightarrow{\tau} (\omega, q, s_c, j \overset{t}{\sim} (D)_{\beta, n, p}) \vdash a \left[ P \right]_{s'} \quad (3.23)$$

Operation \texttt{discard($t$)} destroys timer $t$. The \texttt{discard} operation disables timer $t$, sets instance variable $t$ in the local state of actor $a$ to \textbf{none}, then actor $a$ continues execution of message handler $P$ (Discard).

$$[\text{Discard}] : (\omega, q, s_c, j) \vdash a \left[ \texttt{discard(t)} \cdot P \right]_s \xrightarrow{\tau} (\omega, q, s_c, j \overset{t}{\sim} (D)_{\beta, n, p}) \vdash a \left[ P \right]_{s'} \quad (3.24)$$

A \texttt{trigger} operation performed on a ready timer $t$ results in a \textbf{retriggerring} of $t$. Retriggering can change the identity, timer characteristics, or timeout message of a timer (Retrigger). An implicit
**discard** operation is always performed when a timer is retriggered (Proposition 3.4.1).

\[
[Retrigger] : (\omega, q, s_c) \vdash (a [\text{trigger}(t, f(e), n, p) \cdot P]_s \uparrow t_a(D)_{n,p}) \xrightarrow{\tau} (\omega, q, s_c) \ominus [t'(0), t'_a(D)_{n',p'}] \vdash (a[P]_s' \uparrow t_a(D)_{n,p})
\]

(3.25)

Every implementation of ACube semantics requires mechanisms for restoring discarded timers, and resuming retriggered timers. These mechanisms are applied in situations where message processing is abandoned and pending timer operations must be reversed. One such implementation is described in Section 4.5.2.

**Proposition 3.4 (Timer Laws)**

In this proposition we abbreviate \(\text{trigger}(t, f(e), n, p)\) to \(\text{trigger}(t, f(e))\).

1. An implicit **discard** is always performed when a timer is retriggered (Retrigger). In the following expression \(c^*\) indicates 0 or more control operators.

\[
a[\text{trigger}(t, f(e)) \cdot c^* \cdot \text{trigger}(t, f(e)) \cdot P]_s \equiv a[\text{trigger}(t, f(e)) \cdot c^* \cdot \text{discard}(t) \cdot \text{trigger}(t, f(e)) \cdot P]_s
\]

2. The **discard** operation is idempotent. Once a timer has been discarded, subsequent **discard** operations have no effect.

\[
a[\text{discard}(t) \cdot c^* \cdot \text{discard}(t) \cdot P]_s \equiv a[\text{discard}(t) \cdot c^* \cdot P]_s
\]

**Timer updates.** Timers move between the ready and active state in response to the √ action generated by the master-clock (see Section 2.2.4 on page 35), and by the execution of the **trigger** and **discard** operators by the owner of a timer. Three outcomes are possible when action √ is accepted by a timer \(t\):

1. if \(t\) is a oneshot timer and updating causes \(t\) to expire, a timeout message \(\langle t a \sigma v \rangle\) is sent to actor \(a\) (the owner of \(t\)), then \(t\) is discarded (\(t'(0)\) denotes a discarded timer) (OneShotTimeout),

\[
[\text{OneShotTimeout}] : (a q(D) \uparrow t_a(D)_{1,p}) \xrightarrow{\sqrt{}} (a q(D) \uparrow t'(0)) \mid \langle t a \sigma v \rangle
\]

(3.26)
2. if $t$ is a periodic timer and updating causes $t$ to expire, a timeout message $\langle ta\sigma v \rangle$ is sent to actor $a$ (the owner of $t$), then $t$ is restarted (reloading $n$ from the period established by the most recent trigger operation) and returned to the ready state (PeriodicTimeout), or

$\text{[PeriodicTimeout]} : (a_q(D)_s \uparrow l_a(D)_{1,p}) \xrightarrow{\sqrt{\cdot}} (a_q(D)_s \uparrow l_a(D)_{n,p}) \mid \langle ta\sigma v \rangle$ (3.27)

3. in all other cases, $t$ ticks ($n' = n - 1$) then returns to the ready state ($\sqrt{\cdot}$).

$\text{[\sqrt{\cdot}]} : (a_q(D)_s \uparrow l_a(D)_{n,p}) \xrightarrow{\sqrt{\cdot}} (a_q(D)_s \uparrow l_a(D)_{n',p})$ (3.28)

Timers with the behaviour described in Section 2.2.4 do not require checkpointing support, and timeout messages are never placed on the sequence of pending messages $X$ (OneShotTimeout, PeriodicTimeout). The handling of timeout messages is described in Section 4.3.

### 3.8 Auxiliary Operations

#### 3.8.1 Accessing Local State and Message Parameters

The local state of an actor; denoted $s, s', s'', \ldots$, and the contents of a message; denoted $v, v', v'', \ldots$, are viewed either as scalar instance variables or as a record of instance variables.

The replacement of state variable $x \in S$ with expression $e$ is denoted $\{x \leftarrow e\}$ (LocalState).

$\text{[LocalState]} : (\omega, q, s_c, J) \vdash a[\{x \leftarrow e\}] \cdot P \xrightarrow{x} (\omega, q, s_c, J) \vdash a[P]_{x'}$ (3.29)

Every actor has access to the following set of protected state variables: self which indicates the identity of the active actor, owner which indicates the identity of the owner of the active actor, and current which indicates the current conceptual state of the active actor. An explicit assignment to a protected variable; i.e., $\{\text{self} \leftarrow 0\}$, has no effect; they are only modifiable by an implementation.

Changes made to the local state of an actor are finalized by the commitment operators (see Section 3.4). If an activation finalizes with becomes or terminate, all changes made to the local state will be retained. If an activation finalizes with abort, the local state is rolled back to the condition that existed when the input message was accepted.
The replacement of message parameter \( x \in V \) with expression \( e \) is denoted \( \{ x \leftarrow e \} \) (MessageParameters).

\[
\text{[MessageParameters]} : (\omega, q, s_c, J) \vdash a \left\{ x \leftarrow e \right\} \cdot P_s^{\sigma v} \xrightarrow{\tau} (\omega, q, s_c, J) \vdash a \left[ P_s^{\sigma v} \right]^{\sigma v'} \quad (3.30)
\]

The parameters of an activation message can be modified (for example, \( \{ x \leftarrow e \} \) with \( x \in V \)) before a forward operation is executed (ParamForward).

\[
\text{[ParamForward]} : (\omega, q, s_c, J) \vdash a \left\{ x \leftarrow e \right\} \cdot \text{forward}(b) \cdot P_s^{\sigma v} \xrightarrow{\tau} (\omega, q, s_c, J) \vdash \left( \omega, q, s_c, J \bowtie \langle \star b \sigma v' \rangle \right) \vdash a \left[ P_s^{\sigma v'} \right]^{\sigma v'} \quad (3.31)
\]

### 3.8.2 The Conditional Operator

The conditional operator\(^2\) \( P_t \not< e \not> P_f \) denotes an actor which behaves like \( P_t \) when the expression \( e \) evaluates to true, otherwise it behaves like \( P_f \) (Cond). The delimiters \( \{ \) and \( \} \) are used when \( P_t \) or \( P_f \) consists of multiple operators.

\[
\text{[Cond]} : \begin{array}{c|c|c}
\llbracket e \rrbracket & \text{true} & \llbracket e \rrbracket & \text{false} \\
P_t \not< e \not> P_f & \rightarrow & P_t & \rightarrow \ P_f
\end{array} \quad (3.32)
\]

### 3.9 Exception Handling

Four aspects of ACube semantics are important in the implementation of exception handling strategies in timed actor languages:

1. the semantics of commitment, Equations 3.11–3.13, allow an actor to preserve its local state in situations where it detects errors in its computation,

2. the semantics of acceptance, Equations 3.14 and 3.15, determine when messages will be accepted by an actor,

3. the semantics of transmission, Equations 3.20 and 3.21, determine when messages can be transmitted by an actor, and

4. the semantics of timers, Equations 3.26 and 3.27, allow actors to react to timeouts.

\(^2\)This operator takes the form of the conditional in CSP [41].
[Accept] : \( a \left( \sum_{i \in n-q} rcv(b, \sigma_i(v)) \cdot P_i(x) \right) \) \( \vdash_a (\omega, q, s_c, \{ \}) \) \( \vdash_a P_j(x) \theta_0 \) \( \vdash_a \sigma_j v \) \( \vdash_a (\omega, q, s_c, [\]) \) \( \vdash_a \sigma_j v \) \( \vdash_a 0 \) \( \vdash_a 0 \)

[Reject] : \( a \left( \sum_{i \in n-q} rcv(b, \sigma_i(v)) \cdot P_i(x) \right) \) \( \vdash_a (b' a \sigma_j v) \) \( \vdash_a \sigma_j v \) \( \vdash_a \sigma_j v \) \( \vdash_a (D) \) \( \vdash_a 0 \) \( \vdash_a 0 \)

[Send] : \( (\omega, q, s_c, \)} \( \vdash_a [send(b, f(e)) \cdot P] \) \( \vdash_a (\omega, q, s_c, \{ ab f v \}) \) \( \vdash_a [P] \)

[Forward] : \( (\omega, q, s_c, \)} \( \vdash_a [forward(b) \cdot P] \) \( \vdash_a (\omega, q, s_c, \{ * b \sigma v \}) \) \( \vdash_a [P] \)

[New] : \( (\omega, q, s_c, \)} \( \vdash_a [new(b, C) \cdot P] \) \( \vdash_a (\omega, q, s_c, \{ b (D) \}) \) \( \vdash_a [P] \)

[Trigger] : \( (\omega, q, s_c, \)} \( \vdash_a [trigger(t, f(e), n, p) \cdot P] \) \( \vdash_a (\omega, q, s_c, \{ t (D) \}) \) \( \vdash_a [P] \)

[Discard] : \( (\omega, q, s_c, \)} \( \vdash_a [discard(t) \cdot P] \) \( \vdash_a t (D) \) \( \vdash_a (\omega, q, s_c, \{ t (0) \}) \) \( \vdash_a [P] \)

[Retrigger] : \( (\omega, q, s_c, \)} \( \vdash_a [trigger(t, f(e), n, p) \cdot P] \) \( \vdash_a t (D) \) \( \vdash_a (\omega, q, s_c, \{ t (D) \}) \)

[OneShotTimeout] : \( (\omega, q, s_c, \)} \( \vdash_a [one-shot \cdot P] \) \( \vdash_a (\omega, q, s_c, \{ t (0) \}) \) \( \vdash_a (\omega, q, s_c, \{ t (0) \}) \)

[PeriodicTimeout] : \( (\omega, q, s_c, \)} \( \vdash_a [one-shot \cdot P] \) \( \vdash_a (\omega, q, s_c, \{ t (0) \}) \) \( \vdash_a (\omega, q, s_c, \{ t (0) \}) \)

[LocalState] : \( (\omega, q, s_c, \)} \( \vdash_a [x \leftarrow e \cdot P] \) \( \vdash_a (\omega, q, s_c, \{ x \leftarrow e \}) \)

[MessageParameters] : \( (\omega, q, s_c, \)} \( \vdash_a [x \leftarrow e \cdot P] \) \( \vdash_a (\omega, q, s_c, \{ x \leftarrow e \}) \)

[Becomes] : \( (\omega, q, s_c, \)} \( \vdash_a [becomes(q)] \) \( \vdash_a C \) \( \vdash_a C' \) \( \vdash_a C' \)

[Terminate] : \( (\omega, q, s_c, \)} \( \vdash_a [terminate] \) \( \vdash_a C \) \( \vdash_a C' \) \( \vdash_a C' \)

[Abort] : \( (\omega, q, s_c, \)} \( \vdash_a [abort] \) \( \vdash_a C \) \( \vdash_a C' \) \( \vdash_a C' \)

Figure 3.3: An operational semantics for ACube dynamic operators.
3.10 Example: An ACube Definition for the Vending Machine Configuration

In this section we continue the description of the vending machine configuration introduced in Section 2.4. The configuration analyzed in this chapter (Figure 3.4) includes an additional actor, $m \in \text{Main}$, which bootstraps the vending machine system. The ACube definition for this configuration is shown in Figure 3.5. In this definition we code item ‘a’ as 0, and item ‘b’ as 1.

![Diagram of vending machine configuration](image)

Figure 3.4: A vending machine configuration with bootstrap actor $m \in \text{Main}$.

Since ACube only supports the passing of actor identities it is necessary to recode the input sets of some actor definitions presented in Section 2.4. For example; the send operation $\text{select(‘a’) \Rightarrow v}$ is recoded as $\text{send(v, select_a())}$, and the send operation $\text{coin(5) \Rightarrow v}$ is recoded as $\text{send(v, coin_5())}$. In ACube definitions, state Start is renamed C, where C is the name of actor class C; and becomes(q) is shortened to $q$. 
Main\((u, v)\) def = rcv\((\star, \text{init})\) · new\((u, \text{User})\) · new\((v, \text{Vend})\) · 
send\((u, \text{init}(v))\) · send\((v, \text{init})\) · terminate

User\((t, v)\) def = rcv\((\star, \text{init}(x))\) · \{\(v \leftarrow x\)\} · trigger\((t, \text{noselect()}, 15, \text{oneshot})\) ·
send\((v, \text{select}_b())\) · send\((v, \text{coin}_5())\) · 
send\((v, \text{coin}_5())\) · Dispense

Dispense ::= rcv\((v, \text{dispense}_a())\) · skip · discard\((t)\) · terminate
+ rcv\((v, \text{dispense}_b())\) · skip · discard\((t)\) · terminate

Defaults ::= rcv\((t, \text{noselect}())\) · send\((v, \text{restart}())\) · User

Vend\((x, u)\) def = rcv\((\star, \text{init}())\) · Select

Select ::= rcv\((\star, \text{select}_a())\) · \{\(u \leftarrow \star\)\} · \{\(x \leftarrow 0\)\} · VM0
+ rcv\((\star, \text{select}_b())\) · \{\(u \leftarrow \star\)\} · \{\(x \leftarrow 1\)\} · VM0

VM0 ::= rcv\((u, \text{coin}_5())\) · VM5
+ rcv\((u, \text{coin}_10())\) · \{\ VM10 \} \(\not\prec x = 1 \not\succ\ \{\ send(u, \text{dispense}_a())\cdot Select \} \)

VM5 ::= rcv\((u, \text{coin}_5())\) · \{\ send(u, \text{dispense}_a())\cdot Select \} \(\not\prec x = 0 \not\succ\ \{\ VM10 \} \)
+ rcv\((u, \text{coin}_10())\) · 
\{\ send(u, \text{dispense}_a())\cdot Select \} \(\not\prec x = 0 \not\succ\ \{\ send(u, \text{dispense}_b())\cdot Select \} \)

VM10 ::= rcv\((u, \text{coin}_5())\) · send\((u, \text{dispense}_b())\) · Select
+ rcv\((u, \text{coin}_10())\) · send\((u, \text{dispense}_b())\) · Select

Defaults ::= rcv\((u, \text{restart}())\) · Select

Figure 3.5: ACube definition for a vending machine configuration.

The initial configuration of this system consists of a single actor, \(m \in \text{Main}\), and a pending initialization message for \(m\).

\[ C_0 = \{\{(m, \text{Main})\}, \{\}, [m \cdot \text{init}()]\} \]

Actor \(m\) bootstraps the configuration by creating a vending machine \((v \in \text{Vend})\), and a vending
machine user \((u \in \text{User})\). The \textbf{new} operator places actors \(m, v, \text{and } u\) in a parallel composition. This composition has the form

\[
m[\text{Main}]^{\text{init}}() \mid a([\text{User}]^{\text{init}}()) \mid v([\text{Vend}])
\]

In this composition \(a([Q])\) denotes a ready actor \(a\) in conceptual state \(Q\), and \(a([Q])^{f(v)}\) denotes an active actor \(a\) in conceptual state \(Q\) processing message \(f(v)\).

The send operation \(\text{send}(u, \text{init}(v))\), in the definition of \(\text{Main}\), establishes communication between actors \(u\) and \(v\). Actor \(v\) determines the identity of its user by accepting the message, \(\text{rcv}(*, \text{select}_b())\), and assigning the source identity \((*)\) to state variable \(u\). The interaction between actors \(u\) and \(v\) has previously been summarized in the activation history outlined in Table 2.4.

\[
\begin{align*}
(m[\text{Main}]^{\text{init}}()) & \quad \tau = [\text{New}(2), \text{Send}(2), \text{Terminate}] \\
(m(0) \mid v([\text{Vend}]) \mid u([\text{User}]^{\text{init}}(v))) & \quad \tau = [\text{Trigger}, \text{Send}(4), \text{Becomes}] \\
(u(\text{Dispense}) \mid ^t(Tick) \mid v([\text{Vend}]^{\text{init}}())) & \quad \tau = [\text{Becomes}] \\
(v(\text{Select}) \mid ^t(Tick) \mid u([\text{Dispense}]^{\text{dispense}_b()})) & \quad \tau = [\text{Discard}, \text{Terminate}] \\
(u(\text{Dispense}) \mid ^t(Tick) \mid v([\text{Select}]^{\text{select}_b()})) & \quad \tau = [\text{Send}, \text{Becomes}] \\
(v(\text{Select}) \mid ^t(Tick) \mid u([\text{Dispense}]^{\text{coin}_5()})) & \quad \tau = [\text{Becomes}] \\
(u(\text{Dispense}) \mid ^t(Tick) \mid v([\text{VM0}]^{\text{coin}_5()})) & \quad \tau = [\text{Becomes}] \\
(u(\text{Dispense}) \mid ^t(Tick) \mid v([\text{VM10}]^{\text{coin}_5()})) & \quad \tau = [\text{Becomes}] \\
\end{align*}
\]

Figure 3.6: Vending machine configuration—processing a typical input sequence.

\textbf{A labelled transition system for a vending machine configuration.} For a typical sequence of user input the vending machine configuration can undergo hundreds of transitions. In Figure 3.6 we summarize the results of processing one such input sequence: \(\text{select}_b() \cdot \text{coin}_5() \cdot \text{coin}_5() \cdot \)
coin_5(). To make this summary tractable, we display the state of the system at the commitment points. Since many operators can be executed between commitment points, we label transitions as $\tau = [\vec{o}] \rightarrow$ where $[\vec{o}]$ is any sequence of control operations ending with a commitment operator.

### 3.11 Summary

In this chapter we have described the transitional semantics and algebraic properties of $ACube$, an asynchronous actor algebra. $ACube$ expressions are described in the context of a timed actor configuration $\langle \langle K, J, X \rangle \rangle$. The transitional semantics of $ACube$ expressions allows us to model changes in an actor system, and to describe how actor behaviour is realized. This description defines a timed actor semantics, a structured operational semantics for $ACube$.

Atomic message processing is realized with the aid of an intentions list $I$ which records the sequence of selected dynamic operations executed by an actor between the time it accepts an input message and the time it executes a commitment operation. At the commitment point, an actor can choose to retain its changes, or to rollback the configuration to the state that existed when the input message was accepted.

Our revised model of an actor system, illustrated in Figure 3.7, consists of a timed actor configuration $\langle \langle K, J, X \rangle \rangle$, an intentions list $I$, a master-clock $T$, and the set of $ACube$ dynamic operators. $ACube$ supports the following dynamic operators: $\text{new}(b, C)$ creates a new actor instance $b$ with class behaviour $D_C$, $\text{trigger}(t, f(e), n, p)$ creates and enables a timer $t$ of type $p$ (oneshot or periodic) with duration $n$ and timeout message $f(e)$, $\text{discard}(t)$ destroys timer $t$, $\text{send}(b, f(e))$ sends message $f(e)$ to actor $b$, $\text{forward}(b)$ forwards the current activation message to actor $b$, $\text{becomes}(q)$ finalizes message processing allowing an active actor to enter conceptual state $q$, $\text{terminate}$ finalizes message processing allowing an active actor to self-terminate, $\text{abort}$ aborts message processing allowing the rollback of a configuration to the point where the current input message was accepted, and $\{x \leftarrow e\}$ assigns the value of expression $e$ to local state variable (or message parameter) $x$. 
Many actor algebras [1, 5, 19] provide the **new**, **send**, and **becomes** operations. *ACube* can be distinguished from other actor algebras by: the provision of timer management operations (**trigger** and **discard**), the provision of a class definition facility which structures definitions as communicating finite state machines, and the provision of atomic message processing with **local checkpointing**.

Four aspects of our **timed actor semantics** are important in the implementation of exception handling strategies for a **timed actor language** or a **virtual machine**: the commitment operators allow an actor to preserve its local state in situations where it detects errors in a computation; rules **Accept** and **Reject** indicate when messages will be accepted by an actor; rules **Send** and **Forward** indicate when messages can be transmitted by an actor; and rules **OneShotTimeout** and **PeriodicTimeout** indicate when actors can react to the passage of time.

In Section 3.10 we continue our description of the vending machine system introduced in Section 2.4 by describing how labelled transition systems can be used to illustrate evolution in an actor system.
Chapter 4

A Virtual Machine for a Timed Actor Language

The structured operational semantics of an algebra defines an instruction set for computation on an abstract machine [23, 40, 19]. In this chapter we describe the characteristics of an abstract machine based on ACube timed actor semantics. To accommodate a timed actor language we require a virtual machine which

- simulates an environment suitable for actor execution,

- simplifies the production of an implementation,

- has the ability to adjust the degree of concurrency depending on the demands of the computation, and

- has the ability to distribute a computation among concurrent actors; regardless of locality.

Under ACube timed actor semantics, a local actor system consists of an actor configuration \( \langle \mathcal{X}, \mathcal{J}, \mathcal{X} \rangle \), an intentions list \( \mathcal{I} \), and a master-clock \( T \). The timed actor semantics effectively hides the locality of individual actors, and the nature of the communications medium. In this chapter we
model actor systems as *distributed systems* of the form

\[
\text{System} \defeq \text{Node}_1 \mid \text{Node}_2 \mid \cdots \mid \text{Node}_i \mid \cdots \mid \text{Node}_k
\]  

(4.1)

\[
\text{Node}_i \defeq \left\{ \begin{array}{ll}
\text{Processor}_{i,1} \mid \cdots \mid \text{Processor}_{i,n} & | \quad \text{VM}_{i,1} \mid \cdots \mid \text{VM}_{i,m} | T_i
\end{array} \right.
\]  

(4.2)

\[
\text{VM}_{i,j} \defeq \text{Dispatcher}_{i,j} \mid \{ \text{K}, \text{J}, \text{X} \}_i \mid J_{i,j}
\]  

(4.3)

where each of the \( k \) nodes in the system consists of \( n_i \) processors and \( m_i \) virtual machines. A node \( i \) with \( n_i > 1 \) processors or \( m_i > 1 \) virtual machines is referred to as a *multi-threaded node*. When \( n_i = m_i = 1 \), we have the *single-threaded* case, the organization we examine in this chapter.

In this context, the *global actor configuration* \( \{ \text{K}, \text{J}, \text{X} \} \) described in Chapter 3 is a distributed system represented by the union of \( k \) *local actor configurations* together with their intentions list \( J \) and master-clock \( T \).

\[
\{ \text{K}, \text{J}, \text{X} \} \defeq \bigcup_{i=1}^{k} (\{ \text{K}, \text{J}, \text{X} \}_i \mid J_i \mid T_i)
\]  

(4.4)

Such a simple model does not provide an adequate environment for the description of a kernel for a *timed actor language*; as it omits all consideration of necessary kernel resources; e.g. storage for actor instances, timer instances, and message buffers. In this chapter we illustrate a single-threaded organization (Figure 4.1) for a kernel supporting timed actor semantics where each node has the form

\[
\text{Node} \defeq \text{Processor} \mid T \mid \text{VM} \mid R \mid T \mid \mathcal{B}_N
\]  

(4.5)

\[
\text{VM} \defeq \text{Dispatcher} \mid L \mid \{ \text{K}, \text{J}, \text{X} \} \mid J \mid S \mid \mathcal{B}_V
\]  

(4.6)

In this organization, a node provides the following structures:

- **Processor**—A central processing unit.
- **\( T \)**—A local master-clock.
Figure 4.1: A virtual machine supporting timed actor semantics.

- **VM**—A virtual machine.
- **R**—A system receive buffer; all *nonlocal* inbound messages arrive at the VM through R.
- **T**—A system transmit buffer; all *nonlocal* outbound messages depart the VM through T.
• \( B_N \)—A message buffer pool jointly managed by the VM and the node. This pool provides message buffers for distributed communications between \( R \) and \( X \), and \( X \) and \( T \).

Each virtual machine (VM) provides the following structures:

• Dispatcher—Dispatcher is a specialized actor which manages the kernel. It provides an abstract interface between the kernel and an actor configuration, and an interface between the kernel and the local node.

• \( L \)—Each kernel maintains a scalar logical clock to order the events at that kernel [42, 43, 44]. For any application, these kernel logical clocks can provide a partial ordering of selected dynamic actor operations. \( L \) is used by the logging facility of the VM (see Section C.1).

• \( K \)—The actors list, the set of ready and active actors managed by Dispatcher. The elements of this list are actor instances of the form \( k = (a, q, s) \in K \) (see page 28).

• \( J \)—The timers list, the set of ready and active timers managed by Dispatcher. The elements of this list are timer instances of the form \( j = (t, q, s) \in J \) (see page 30).

• \( X \)—The sequence of pending messages managed by Dispatcher. \( X \) is partitioned into three sequences: \( X_L \) is provided to support local messaging; and \( X_I \) (inbound messages) and \( X_O \) (outbound messages) are provided to support distributed messaging. The elements in these sequences are references to messages allocated from the kernel message buffer pool \( B_V \).

• \( I \)—The intentions list is provided to support atomic message processing. The elements of this list are references to actor instances, timer instances, and messages created by the active actor. This list is processed each time the active actor reaches a commitment point.

• \( S \)—The state list. This list is partitioned into two sublists: \( S_K \) accommodates the local state of actors in \( K \), and \( S_J \) accommodates the local state of timers in \( J \).

• \( B_V \)—A message buffer pool managed by the VM. The elements of this list accommodate messages of the form \( \alpha = (\star, a, \sigma, v) \in X \) (see page 29). This pool provides message buffers for communications managed by the kernel.
Functions head(\(X\)) and tail(\(X\)). The structure of the virtual machine illustrated in Figure 4.1 alters the semantics of functions head(\(X\)) and tail(\(X\)) (see Section 2.2.3 on page 32).

\[
\text{head}(X) = \text{head}(X_L) \lor \text{head}(X_I) \quad (4.7)
\]

\[
\text{tail}(X) = \text{tail}(X_L) \lor \text{tail}(X_O) \quad (4.8)
\]

Equation 4.7 indicates that head(\(X\)) resolves to either head(\(X_L\)) or head(\(X_I\)), for instance when Dispatcher schedules the next message for processing; and Equation 4.8 indicates that tail(\(X\)) resolves to either tail(\(X_L\)) or tail(\(X_O\)), for instance when the VM appends the next output message to the list of pending messages (\(X\)).

Outline. The goal of this chapter is to describe a kernel (VM or virtual machine) to accommodate a timed actor language, and show how this kernel can be derived from the timed actor semantics presented in Chapter 3. We describe the:

- Models of execution—In Section 4.1 we describe the model of actor execution supported by the kernel.

- Actor template—In our approach, the unit of actor execution is referred to as the actor template (Section 4.2).

- Kernel—Our description of the kernel focuses on three components.
  - A dispatcher for a VM implementing timed actor semantics—In Section 4.3 we describe the role of Dispatcher in our kernel.
  - Dispatcher-actor interface—In Section 4.4 we describe the interface used by Dispatcher to activate actors, process the intentions list, and handle run-time exceptions.
  - Actor-kernel interface—In Section 4.5 we examine how actors interact with the kernel, and how VM operations are derived from our timed actor semantics.

- Kernel syntactic domains—In Section B.1 we provide a summary of the key syntactic domains used to describe the kernel.
• VM supporting timed actor semantics—Our description of the kernel and VM operations is based on our implementation of the VM provided as pseudocode in Section B.2.

In this chapter the terms virtual machine (VM), ACube VM, run-time kernel, and kernel are equivalent.

4.1 Models of Execution

Formal models of distributed systems distinguish synchronous from asynchronous models of execution. A synchronous model is one in which nodes are driven by a global clock which generates time intervals of fixed duration, actions are atomic executing in zero-time, and the delay that a message experiences is strictly less than the interval of the global clock [45, 46, 47]. An asynchronous model is one in which each node is driven by a local clock, actions are not atomic and execute in a finite but unpredictable amount of time, and the delay that a message experiences is finite but unbounded [46, 47, 48].

For a practitioner neither of these two models characterizes the nature of execution and communication in the Timed Actor model. The model of execution and communication in the Timed Actor model is fundamentally asynchronous, with the following qualifications:

• Each node has access to a local master-clock, $T$.

• Actors only react to the arrival of messages. The reception of a message is subject to certain acceptance conditions; once accepted the actor handles the message deterministically.

• The communications medium is lossy. Since a timed actor language may provide guarded commands, a practitioner is unable to distinguish a message dropped by the communications medium from one dropped by the implementation. All exceptions are mapped by the kernel or compiler into lost messages.

• The processing of a message takes a finite amount of time ($t_e$). We take the following precautions to insure that message processing has low latency:
○ Actors are single-threaded, they can only process one message at a time.
○ Message processing is atomic, in the following sense. Like the synchronous model of execution, actions are indivisible. In addition, message processing is atomic, so the message will either be completely processed by the active actor, or it will be abandoned and the state of the active actor is returned to its most recent checkpoint. To implement this type of message processing it is important that the active actor not be preempted.
○ At the language level a practitioner should accommodate high-latency message handlers by dividing them into a sequence of low-latency message handlers activated by self messages.

• The delay a message experiences is finite ($t_e$), but bounded when guarded by a timer.

• Message handlers should be designed so that $t_e \ll t_c$, however, neither the kernel nor the timed actor language enforce this relationship. This relationship can simplify the implementation of run-time monitors and the interpretation of monitoring results.

4.2 The Actor Template

The ACube definition for actor class $C$ describes the behaviour of all actor instances of class $C$. Every application requires an application class table, which contains an entry for each actor class defined in an ACube specification. Each class table entry provides two pieces of information to the kernel: $\text{process}_C$, the address of an actor template for executing actor instances belonging to class $C$; and $\text{constructor}_C$, the address of a class constructor for initializing an actor instance belonging to class $C$.

```
ClassTable = \{ process_1 | constructor_1, \ldots, process_n | constructor_n \}
```
For example, application DBUFF described in Listings D.2–D.3 would require class table entries for classes Buffer, Producer, Consumer, and Main.

<table>
<thead>
<tr>
<th>Class</th>
<th>Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>BufferProcess</td>
<td>BufferConstructor</td>
</tr>
<tr>
<td>ProducerProcess</td>
<td>ProducerConstructor</td>
</tr>
<tr>
<td>ConsumerProcess</td>
<td>ConsumerConstructor</td>
</tr>
<tr>
<td>MainProcess</td>
<td>MainConstructor</td>
</tr>
</tbody>
</table>

**Actor template.** The actor template for instances of actor class C contains the VM instructions to implement the behaviour of class C. For example, the ACube definition of actor class Buffer in Figure 4.2 can be manually translated into the actor template illustrated in Listing 4.1.

\[
\text{Buffer}(p, c, n = 20) \overset{\text{def}}{=} \text{rcv}(\ast, \text{init}(x, y)) \cdot \{ p \leftarrow x \} \cdot \{ c \leftarrow y \} \cdot \text{initQueue}(n) \cdot \text{Empty} \\
\ldots
\]

\[
\text{Empty}() \overset{\text{def}}{=} \text{rcv}(p, \text{put}(x)) \cdot \text{enQueue}(x) \cdot \{ \text{Full} \} \not\leq \text{isFull()} = 1 \not\geq \{ \text{PartFull} \} \\
+ \text{rcv}(c, \text{get}()) \cdot \text{send}(c, \text{underflow}()) \cdot \text{Empty} \\
\ldots
\]

Figure 4.2: ACube definition for \( b \in \text{Buffer} \).

An actor template generated by a compiler for a timed actor language utilizes the VM instructions described in Section 4.5 and summarized in Table 4.4. The behaviour of these VM instructions is equivalent to their ACube counterparts. The functions used in the description of an actor template and the kernel are summarized in Table B.2.

When an actor \( a \in C \) is activated, Dispatcher performs a context-switch yielding the thread-of-control to actor \( a \) at the address of the actor template for class C. An actor template requires two actual parameters; \( s \), a reference to the local state of actor \( a \); and \( m \), a reference to the next input message for actor \( a \). Two scoping levels are apparent in Listing 4.1; \( s \) provides access to the local state of \( a \), and \( v \) provides access to the message contents of \( m \).
The body of an actor template employs nested case statements to select the correct message handler \(d\) in the current conceptual state \(q\). The kernel raises exception NOHANDLER when actor \(a\) receives a message \(\sigma \notin \Sigma_{a,q}\).
Guarded input in a receive statement, for example \texttt{rcv}(p,\texttt{put}(x))\texttt{,} is translated into a selection statement of the form:

\[
\text{if } \text{source}(m) = s.p \text{ then } \langle \text{body of put} \rangle
\]

When an active actor reaches a commitment point, it yields the thread-of-control to Dispatcher, returning two arguments; \( r \in R \), the commitment action; and \( q' \in Q \), the next conceptual state of actor \( a \).

\textbf{Class constructor.} A compiler for \( ACube \) must generate a constructor for each class \( C \) in an \( ACube \) specification. The class constructor contains an assignment operation, denoted \( x \leftarrow e \), for each local state initializer of the form \( x = e \) in a class definition. In the definition of class \( Buffer \) initializer \( n = 20 \) generates the assignment \( s.n \leftarrow 20 \) in class constructor \( BufferConstructor \).

\textbf{Message acceptance semantics.} The design of an actor template enforces the \emph{acceptance conditions} described in Definition 3.3 on page 59. When actor \( a \) receives a message \( \sigma \notin \Sigma_{a,q} \) it aborts message processing, enforcing condition \( C_1 \). When the identity of the sender of message \( m \) does not match identity \( b \) of an input guard the actor commits, enforcing condition \( C_2 \).

\section{4.3 A Dispatcher for a VM Implementing Timed Actor Semantics}

In our kernel, Dispatcher is a specialized actor responsible for the management of the VM. The \emph{actor template} for Dispatcher is illustrated in Listing B.1 (DspProcess), and the behaviour of Dispatcher is captured by the finite state machine in Figure 4.3 and Table 4.1.

At initialization Dispatcher starts the event logging facility (\texttt{StartLog()}\texttt{)} then begins a polling service. The polling service terminates whenever an actor invokes operation \texttt{cexit()} to terminate the application. Once the polling service has terminated the logging facility (\texttt{EndLog()}\texttt{)} prepares an \emph{event log} of all monitored activity.
Figure 4.3: Dispatcher, a finite state machine model of dispatcher behaviour.

While executing the polling service Dispatcher activates ready actors in three situations: when a timer $t \in J$ has expired and a timeout message is to be delivered to its owner, when an input message is available on sequence $X_I$, or when input message is available on sequence $X_L$. The polling service imposes an implicit priority on message processing, that is, if messages are simultaneously available from all three sources, Dispatcher handles them in the following order: $J \rightarrow X_I \rightarrow X_L$. Another characteristic of the polling service is that after work has been performed in states $S_2$ or $S_3$, the service is restarted in state $S_1$. This behaviour insures that the dispatch latency of timeout messages is minimized. Dispatching is performed by kernel operation `vmdispatch` (see Section 4.4.1).

Table 4.1: States and transitions describing the behaviour of Dispatcher.

<table>
<thead>
<tr>
<th>State</th>
<th>State Description</th>
<th>Transition</th>
<th>Transition Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>Check $J$</td>
<td>$S_1 \rightarrow S_2$</td>
<td>Processing of $J$ complete, check $X_I$.</td>
</tr>
<tr>
<td>$S_2$</td>
<td>Check $X_I$</td>
<td>$S_2 \rightarrow S_1$</td>
<td>Move $R$ to $X_I$, dispatch head($X_I$), restart poll.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$S_2 \rightarrow S_3$</td>
<td>$X_I$ empty, or not schedulable, check $X_O$.</td>
</tr>
<tr>
<td>$S_3$</td>
<td>Check $X_O$</td>
<td>$S_3 \rightarrow S_1$</td>
<td>Move $X_O$ to $J$, restart poll.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$S_3 \rightarrow S_4$</td>
<td>$X_O$ empty, check $X_L$.</td>
</tr>
<tr>
<td>$S_4$</td>
<td>Check $X_L$</td>
<td>$S_4 \rightarrow S_1$</td>
<td>Dispatch head($X_L$), or $X_L$ is empty, restart poll.</td>
</tr>
</tbody>
</table>
The polling service performs four main tasks:

1. State $S_1$, Update timer list $\mathcal{J}$ — At the beginning of each poll the timer list $\mathcal{J}$ is scanned for expired timers. Algorithm CheckTimers (Listing B.1) describes how our kernel manages timers. If an expired timer $t$ is encountered during the scan, one of two outcomes are possible. If $a \notin \text{actors}(\mathcal{K})$, where $a$ is the owner($t$), then $a$ has terminated and Dispatcher discards timer $t$. If $a \in \text{actors}(\mathcal{K})$, message($t$) is dispatched directly to actor $a$. Timeout messages are never inserted into $\mathcal{X}_L$.

When control returns to Dispatcher, timer $t$ is either discarded (oneshot), or retrigged (periodic); a behaviour derived from the semantics of timeout described in Section 3.7, and summarized in transition rules OneShotTimeout and PeriodicTimeout on page 63.

The dispatcher remains in state $S_1$ until all expired timers are processed. When timer updates are complete, the state machine transitions to state $S_2$.

2. State $S_2$, Check inbound message sequence $\mathcal{X}_I$ — Algorithm CheckInbound (Listing B.1) describes how our kernel manages inbound messages. In state $S_2$ Dispatcher drains system receive buffer $\mathcal{R}$ into inbound message sequence $\mathcal{X}_I$, an action denoted by $\mathcal{X}_I \leftarrow \mathcal{R}$. Each time a message is moved from $\mathcal{R}$ to $\mathcal{X}_I$ Dispatcher updates the kernel logical clock ($\mathcal{L}$). If $\mathcal{L}$ is the current value of the kernel logical clock and $\text{clock}($message$)$ extracts the timestamp from input message $m$, an updated kernel logical clock is computed as

$$\mathcal{L} \leftarrow \text{inc}(\max(\mathcal{L}, \text{clock}(m)))$$

Dispatcher checks if $\mathcal{X}_I$ is schedulable each time algorithm CheckInbound is invoked. Function Schedule (Listing B.1) alternates the dispatch of input messages between sequences $\mathcal{X}_I$ and $\mathcal{X}_L$. This leads to fair message processing and avoids the possibility of starving local actors communicating through $\mathcal{X}_L$.

When the processing of $\mathcal{X}_I$ is not schedulable the dispatcher transitions to state $S_3$. When the processing of $\mathcal{X}_I$ is schedulable, and $\mathcal{X}_I$ is not empty, message $m = \text{head}(\mathcal{X}_I)$ will be
dispatched to target actor $a = \text{dest}(m)$. When control returns to Dispatcher the poll is restarted in state $S_1$.

3. State $S_3$, Check outbound message sequence $X_O$ — Algorithm CheckOutbound (Listing B.2) describes how our kernel manages outbound messages. If $X_O$ is empty the dispatcher transitions to state $S_4$. If $X_O$ is not empty the dispatcher drains $X_O$ by moving all messages in $X_O$ to system transmit buffer $T$, an action denoted by $T \leftarrow X_O$. As each message $m$ is moved from $X_O$ to $T$, the kernel logical clock ($L$) is incremented, and written as a timestamp to field clock of message $m$. When $X_O$ is drained the poll is restarted in state $S_1$.

4. State $S_4$, Check local message sequence $X_L$ — Algorithm CheckLocal (Listing B.2) describes how our kernel manages local messages. If $X_L$ is empty the dispatcher transitions to state $S_1$. If $X_L$ is not empty, message $m = \text{head}(X_L)$ will be dispatched to target actor $a = \text{dest}(m)$. When control returns to Dispatcher the poll is restarted in state $S_1$.

Dispatcher implements a time-action tree model of execution commonly associated with timed process algebras [49, 26]. In this model the polling service alternates between advancing time and performing instantaneous actor execution.

### 4.4 The Dispatcher-Actor Interface

In this section we examine how Dispatcher interacts with actors. An interface between Dispatcher and the actors of a local configuration provides support for dispatching, processing the intentions list, and exception handling (see Table 4.2). The syntactic domains used in this interface are described in Section B.1.

At each node participating in a distributed application, an instance of Dispatcher shares a single-thread of control with the active actor. Dispatcher uses operation `vmdispatch` to activate a ready actor (see Section 4.4.1). When an active actor executes a commitment operation, control is returned to Dispatcher which invokes operation `vmprocess` to process the intentions list $I$ (see Section 4.4.2). Exceptions raised by the kernel or active actor are reported using operation `vmnotify`
Table 4.2: A Dispatcher-actor interface.

<table>
<thead>
<tr>
<th>Description</th>
<th>Abstract Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispatch message</td>
<td>( \text{vmdispatch} : A \times X &amp; \mapsto \emptyset )</td>
</tr>
<tr>
<td>Process ( I )</td>
<td>( \text{vmprocess} : L \times U \mapsto \emptyset )</td>
</tr>
<tr>
<td>Checkpoint local state</td>
<td>( \text{vmcheckpoint} : A \mapsto \emptyset )</td>
</tr>
<tr>
<td>Enter active state</td>
<td>( \text{vmactive} : A \times S&amp; \times X &amp; \mapsto \emptyset )</td>
</tr>
<tr>
<td>Report exception</td>
<td>( \text{vmnotify} : A \times E \times Q \times \cdots \mapsto \emptyset )</td>
</tr>
<tr>
<td><strong>Templates</strong></td>
<td></td>
</tr>
<tr>
<td>Invoke actor template</td>
<td>( \text{aProcess} : S&amp; \times X &amp; \mapsto R \times Q )</td>
</tr>
<tr>
<td>Invoke class constructor</td>
<td>( \text{aConstructor} : S&amp; \mapsto \emptyset )</td>
</tr>
</tbody>
</table>

### 4.4.1 `vmdispatch`—An Operation to Dispatch a Ready Actor

Dispatcher invokes `vmdispatch` during a poll if, a timer \( t \) expires, an input message \( m \) exists on \( X_I \), or an input message \( m \) exists on \( X_L \). Scheduling of dispatches is described in Section 4.3. Transition rule Deliver describes the removal of a message from \( X_I \) or \( X_L \).

\[
[\text{Deliver}] : \langle a b \sigma v \rangle \xrightarrow{ab\sigma v} \langle 0 \rangle \quad (4.9)
\]

When `vmdispatch` is invoked the following actions are taken (see Listing B.2).

1. If destination actor \( a \) has terminated, the input message is discarded and control is returned to Dispatcher.

2. If destination actor \( a \) exists, its local state is checkpointed. This ensures that the original local state can be recovered should actor \( a \) decide to abort message processing.

3. The intentions list \( I \) is reset, the now instance variable of actor \( a \) is assigned the current value of \( \mathbb{T} \), then Dispatcher releases the thread of control to actor \( a \) by invoking its actor template.

4. When actor \( a \) reaches a commitment point, the thread of control is returned to Dispatcher. As part of the context switch, the actor template returns a commitment action (one of `becomes`,
abort, or terminate) which determines how the intentions list $I$ will be processed (see Section 4.4.2).

5. Dispatcher returns to the polling loop.

Message acceptance semantics. The behaviour of operation vmdispatch is derived from the semantics for message acceptance described in Section 3.5.1, and summarized in transition rules Accept and Reject on page 59.

4.4.2 vmprocess—An Operation for Processing an Intentions List

Operation vmprocess is invoked by Dispatcher to finalize changes to the local configuration made during the execution of actor $a$ (see Listing B.3). The execution history of actor $a$ is recorded on the intentions list $I$. The nature of processing is determined by the configuration update action $u$.

Action commit. If actor $a$ executes commitment operator becomes or terminate, the kernel requests configuration update action commit to process the intentions list. The processing performed by vmprocess depends on the type of the intentions list entry:

- $K\&$—If the entry is a reference to actor instance $k$ with identity $b$, Dispatcher invokes $\text{vmnew}'(b, \text{commit})$ which executes the class constructor for $b$.

- $Jt\&$—If the entry is a reference to timer instance $j$ involving the creation of timer $t$, Dispatcher invokes $\text{vmtrigger}'(j, j', t, \text{commit})$ which enables timer $t$. If this is a retriggering of an existing timer, the original timer instance is discarded.

- $Jd\&$—If the entry is a reference to timer instance $j$ involving the destruction of timer $t$, Dispatcher invokes $\text{vmdiscard}'(t, \text{commit})$ which destroys timer $t$.

- $X\&$—If the entry is a reference to message $m$, Dispatcher invokes $\text{vmsend}'(\text{source}(m), m, \text{commit})$ which appends message $m$ to either $X_L$ or $X_O$ depending on the value of node(dest($m$)).
If actor $a$ terminates, the kernel recovers all storage allocated to actor instance $k$. This has the effect of removing actor $a$ from the local configuration.

**Action rollback.** If actor $a$ executes commitment operator `abort`, the kernel requests configuration update action `rollback` to process the intentions list. As part of the rollback, the private local state of actor $a$ is restored from its checkpoint. The processing performed by `vmprocess` depends on the type of the intentions list entry:

- $K$&—If the entry is a reference to actor instance $k$ with identity $b$, Dispatcher invokes  
  \[ \text{vmnew}'(b, \text{rollback}) \]  which destroys actor instance $b$.

- $J_t$&—If the entry is a reference to timer instance $j$ involving the creation of timer $t$, Dispatcher invokes  
  \[ \text{vmttrigger}'(j, j', t, \text{rollback}) \]  which destroys timer instance $j$. If this was an attempt to retrigger an existing timer, the original timer will be restored.

- $J_d$&—If the entry is a reference to timer instance $j$ involving the destruction of timer $t$, Dispatcher invokes  
  \[ \text{vmdiscard}'(t, \text{rollback}) \]  which restores timer instance $j$.

- $X$&—If the entry is a reference to message $m$, Dispatcher invokes  
  \[ \text{vmsend}'(\text{source}(m), m, \text{rollback}) \]  which returns $m$ to buffer pool $B_V$.

**Configuration update semantics.** The behaviour of operation `vmprocess` is derived from the configuration update semantics described in Section 2.2.3, and summarized in Equations 2.11–2.13 on page 34.

### 4.4.3 `vmnotify`—Reporting Exceptions

Exceptions can be raised either explicitly by the kernel or implicitly by an active actor when input messages are dispatched, when input messages are accepted, when messages are transmitted, when resources become exhausted, and when the current conceptual state of an active actor becomes undefined. All exceptions raised by the kernel or active actor are handled with operation `vmnotify` (Table 4.3).
When an application is monitored (see Chapter 6), all exceptions handled by operation `vmnotify` are logged for analysis [13].

Table 4.3: Exceptions reported by operation `vmnotify`.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Where Raised ¹</th>
<th>How Handled</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOTARGET</td>
<td>CheckTimers,</td>
<td><code>vmabort</code></td>
<td>At dispatch actor $a$ does not exist ($\langle ba \sigma v \rangle \land a = \text{none}$).</td>
</tr>
<tr>
<td></td>
<td>CheckInbound,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CheckLocal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOHANDLER</td>
<td>process$_C$</td>
<td><code>vmabort</code></td>
<td>No message handler $m$ in state $q$ ($m \not\in \Sigma_{a,q}$).</td>
</tr>
<tr>
<td>MGFAILED</td>
<td>process$_C$</td>
<td><code>vmcommit</code></td>
<td>Message guard failed; $\text{rcv}(e, m(x))$ where $[e] = \text{false}$.</td>
</tr>
<tr>
<td>TGFAILED</td>
<td>process$_C$</td>
<td><code>skip</code></td>
<td>Type guard associated with transmission $m(e)$ has failed $m(e) \Rightarrow (P) a \land a \not\in P$.</td>
</tr>
<tr>
<td>NORESOURCE</td>
<td><code>vmnew</code>, <code>vmtrigger</code></td>
<td><code>skip</code></td>
<td>Actor or timer resources are exhausted.</td>
</tr>
<tr>
<td>NORESOURCE</td>
<td><code>vmsend</code>, <code>vmforward</code></td>
<td><code>halt</code></td>
<td>Message buffers are exhausted.</td>
</tr>
<tr>
<td>NOSTATE</td>
<td>process$_C$</td>
<td><code>vmterminate</code></td>
<td>The active actor $a$ is in an undefined conceptual state ($q \not\in Q_a$).</td>
</tr>
</tbody>
</table>

Exception **NOTARGET.** Exception NOTARGET occurs during dispatch when the identity of the target actor $a$ in message $\langle ba \sigma v \rangle$ evaluates to `none`, or when target actor $a$ does not exist. The kernel aborts the dispatch of any message with these properties.

Exception **NOHANDLER.** Exception NOHANDLER occurs when the active actor is unable to process the current input message $m$ in the current conceptual state, i.e., $m \in \Sigma_a$ but $m \not\in \Sigma_{a,q}$. The most common cause of this exception is the absence of a definition for message handler $m$ in the current conceptual state (a possible protocol error). When this exception occurs the active actor aborts

¹process$_C$ is the actor template for actor instances belonging to actor class $C$. 
(vmabort).

**Exception MGFAILED.** Exception MGFAILED occurs when the active actor is unable to accept its next input message because a message guard has failed. For example, the receive operation \( \text{rcv}(p, m(x)) \) will fail if the identity of the source actor \( b \) in activation message \( \langle ba v \rangle \) does not match \( p \). If a message guard fails the active actor commits (vmcommit).

Some timed actor languages support a generalized form of message guards, \( \text{rcv}(e, m(x)) \), where \( e \) is an assertion. This type of message guard fails when \( [e] \) evaluates to \text{false}.

**Exception TGFAILED.** In a timed actor language supporting type guards (see Section 5.2.1), the failure of a type guard raises exception TGFAILED. For example, in the following definition

```plaintext
actor class R {
    actor p;
    int x;

    S1:
    msg init() { f(0,1) ⇒ (P)p; x += 1; }
}
```

if \( p \not\in P \), the kernel does not execute statement \( f(0,1) \Rightarrow p \), but skips to statement \( x += 1 \). Output messages are dropped when type guard failures occur.

**Exception NORESOURCE.** Actors, timers, and messages require resources managed by the kernel. The kernel skips \text{vmnew} and \text{vmtrig}ger requests it is unable to complete, and sets the associated actor and timer variable to \text{none}. Execution is halted when message buffers are exhausted.

**Exception NOSTATE.** An actor \( a \) is unable to accept its next message when its current conceptual state is undefined (\( q \not\in Q_a \)). This can occur when the state of an actor becomes corrupt. The kernel terminates any actor which is activated with an undefined conceptual state.
4.5 The Actor-Kernel Interface

In this section we examine how actors interact with the kernel. An interface providing actors access to kernel implementations of the ACube operators described in Chapter 3, is summarized in Table 4.4. The syntactic domains used in this interface are described in Section B.1.

To simulate an environment for actor execution it is necessary to divide some interface operations into two parts. For example, the interface operation which implements the ACube new operator is split into two VM operations denoted vmnew and vmnew'. Operation vmnew is an access point to the kernel used to request all necessary resources and initializations to create a new actor instance. Operation vmnew' is a deferred operation invoked by Dispatcher when it encounters a reference to an actor instance (K&) while processing intentions list. All operations of the form vmop' are designed to deal with the configuration update actions commit and rollback.

Table 4.4: Actor-kernel interface.

<table>
<thead>
<tr>
<th>Description</th>
<th>ACube Operator</th>
<th>VM Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate Operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create actor</td>
<td>new(a, C)</td>
<td>vmnew: A × C → A</td>
</tr>
<tr>
<td>Create oneshot timer</td>
<td>trigger(t, m(e), i, oneshot)</td>
<td>vmtrigger: A × T × M × N × oneshot → T × X &amp;</td>
</tr>
<tr>
<td>Create periodic timer</td>
<td>trigger(t, m(e), i, periodic)</td>
<td>vmtrigger: A × T × M × N × periodic → T × X &amp;</td>
</tr>
<tr>
<td>Discard timer</td>
<td>discard(t)</td>
<td>vmdiscard: A × T → V</td>
</tr>
<tr>
<td>Send message</td>
<td>send(a, m(e))</td>
<td>vmsend: A × A × M → X &amp;</td>
</tr>
<tr>
<td>Forward message</td>
<td>forward(a)</td>
<td>vmforward: X &amp; A × A × M → V</td>
</tr>
<tr>
<td>Change conceptual state</td>
<td>becomes(q)</td>
<td>vmbecomes: Q → R × Q</td>
</tr>
<tr>
<td>Discard actor</td>
<td>terminate</td>
<td>vmtterminate: V → R × Q</td>
</tr>
<tr>
<td>Abort</td>
<td>abort</td>
<td>vmabort: V → R × Q</td>
</tr>
<tr>
<td>Assignment</td>
<td>{ x ← e }</td>
<td>s.x ← e, v.x ← e, x ← e</td>
</tr>
</tbody>
</table>

| Deferred Operations       |                |                                  |
| Create actor              | vmnew': A × U → V |
| Create timer              | vmttrigger': J × T × U → V |
| Discard timer             | vmdiscard': A × U → V   |
| Message transmission      | vmsend': A × X × U → V   |

In this section we extend the timed actor semantics described in Chapter 3 by illustrating the effect of selected VM instructions on a local configuration at a commitment point implemented by
operator \texttt{vmcommit} (an abbreviation for \texttt{vmbecomes(current)}). Transition rules in this section take the form

\[
[Vmlocal]: \quad (\omega, q, s_c, J) \vdash_a \{s.x \leftarrow e\} \cdot \texttt{vmcommit} \xrightarrow{\tau} (\omega, q, s_c, J) \vdash_a [\texttt{vmcommit}]_{s'}
\]

where, the premise describes the effect of a VM instruction (in this case an assignment \(\{s.x \leftarrow e\}\)) on the active actor and the intentions list \(J\); and the conclusion describes the effect of the VM instruction on the local configuration when commitment operator \texttt{vmcommit} is executed. In these transition rules, \((\omega, q, s_c, J)\) is a checkpoint, and \(\{K, J, \text{tail}(X)\}\) is the state of a local configuration before commitment.

### 4.5.1 Actor Creation Operation

Actors extend the notion of objects to concurrent computation. In ACube semantics an actor consists of a private local state, a set of message handlers, zero or more private local timers, a single thread of control, a globally unique identity, and a definition of its behaviour.

On entering operation \texttt{vmnew(a,C)} the kernel performs the following sequence of operations:

1. Create a fresh actor identity \(b \leftarrow \text{fresh(node(a),C)}\).
2. Allocate an actor instance \(k \in \mathcal{K}\), and two local state blocks. Two local state blocks are required to implement local checkpointing.
3. Initialize actor instance \(k\). This includes setting the initial conceptual state \texttt{current} to \texttt{Start}, \texttt{owner} \leftarrow a, and \texttt{self} \leftarrow b.
4. Add a reference to actor instance \(k\) to the intentions list \(J\) and return the fresh identity \(b\) to the active actor.

Dispatcher invokes \texttt{vmnew'} when it encounters a reference to actor instance \(k\) \((K&c)\) while processing the intentions list \(J\). If the active actor commits, Dispatcher executes the class constructor to initialize the user-defined instance variables. If the active actor aborts message processing, Dispatcher recovers all resources allocated to actor instance \(k\).
Actor creation semantics. The behaviour of operation `vmnew` is derived from the semantics for actor creation described in Sections 3.6, and summarized in transition rule `New` on page 62.

Transition rule `Vmnew` illustrates the outcome of processing a single `vmnew` instruction; actor instance $K_b$ is added to list of actors $\mathcal{K}$, and a fresh actor $b$ and its owner $a$ are composed. Actor identities can be created in any scope; assigning the result of `vmnew` to a state variable makes the identity persistent across activations.

\[
\text{[Vmnew]} : \\
\begin{array}{c}
\left(\omega, q, s, \mathcal{J}\right) \vdash \begin{array}{c}
\text{s, b} \leftarrow \text{vmnew}(a, C) \cdot \text{vmcommit} \\
\text{vmcommit}
\end{array} \\
\rightarrow \\
\left(\omega, q, s, \mathcal{J}^\prime, b \leftarrow (D_C)_{s_0} \right) \vdash \begin{array}{c}
\text{vmcommit}
\end{array} \\
\end{array}
\]

The pseudocode implementation of `vmnew` is shown in Listings B.4.

4.5.2 Timer Management Operations

In `ACube` semantics, timers are specialized actors with a private local state, a set of methods, a single thread of control, and a globally unique identity. All timers are instances of class `Timer`. The behaviour of timers is described in Section 2.2.4.

Operation `vmtrigger`—Create a new timer instance. On entering operation `vmtrigger(a, t, d, i, p)` the kernel performs the following sequence of operations:

1. Since we support the retrigerring of timers, if the timer $t \neq \text{none}$ we store a reference for timer instance $j$ to $j'$.
2. Create a fresh timer identity $t \leftarrow \text{fresh}(\text{node}(a), \text{Timer})$.
3. Allocate a fresh timer instance $j \in \mathcal{J}$, a local state block, and a message buffer $m \in B_V$.
4. Initialize timer instance $j$. This includes setting the initial conceptual state `current` to `Start`, `owner` $\leftarrow a$, `self` $\leftarrow t$, `period` $\leftarrow i$, and `type` $\leftarrow p$. The flag `enabled` is set to `false` to indicate the timer is not running.
5. Initialize message buffer $m$. This includes setting `source` $\leftarrow t$, `dest` $\leftarrow a$, and `mid` $\leftarrow d$. 
6. Add a reference to timer instance \( j \) to the intentions list \( I \) and return a fresh timer identity \( t \), and message reference \( m \) to the active actor.

Dispatcher invokes \texttt{vmtrigger} when it encounters a reference to timer instance \( j \) \((Jt&)\) while processing the intentions list \( I \). If the active actor commits, and we are retriggering \( t \), Dispatcher discards timer instance \( j' \); otherwise, the kernel sets counter to \( i \), and flag enabled to \texttt{true} indicating that timer \( t \) is enabled. If the active actor aborts message processing, Dispatcher recovers all resources allocated to timer instance \( j \) then restores timer \( t \) from timer instance \( j' \).

**Timer creation semantics.** The behaviour of operation \texttt{vmtrigger} is derived from the semantics for timer creation described in Sections 3.7, and summarized in transition rule Trigger on page 62.

Transition rule \texttt{Vmtrigger} illustrates the outcome of processing a single \texttt{vmtrigger} instruction; timer instance \( J_t \) is added to list of timers \( J \), and a fresh timer \( t \) and its owner \( a \) form a association. Fresh timer identities can only be assigned to state variables. The marshalling of parameters for timeout message \( m \) is handled by the kernel.

\[
\text{Transition rule } \texttt{Vmtrigger} : \\
\frac{(\omega, q,s_c,J) \vdash a[s.t,m \leftarrow \text{vmtrigger}(a,t,f,i,p) \cdot \text{vmcommit}]_s \quad \tau}{(\omega, q,s_c,J \leftarrow t(D)_{i,p}) \vdash a[\text{vmcommit}]_s' \quad \langle \langle K, J \cup \{J_t\}, \text{tail}(X) \rangle \rangle \vdash (a(D)_{s'} \upharpoonright t(D)_{i,p})}
\]

Transition rule \texttt{Vmretrigger} illustrates the outcome of retriggering an existing timer; timer instance \( J_t \) is removed from list of timers \( J \), timer instance \( J_t' \) is added to list of timers \( J \), and a fresh timer \( t' \), dead timer \( t \), and their owner \( a \) form a association.

\[
\text{Transition rule } \texttt{Vmretrigger} : \\
\frac{(\omega, q,s_c,J) \vdash (a[s.t',m \leftarrow \text{vmtrigger}(a,t,f,i,p) \cdot \text{vmcommit}]_s \quad \tau}{(\omega, q,s_c,J \leftarrow [t'(0), t'(D)_{i,p}]) \vdash (a[\text{vmcommit}]_s' \upharpoonright t'(D)_{i,p}) \quad \langle \langle K, J - \{J_t\} \cup \{J_t'\}, \text{tail}(X) \rangle \rangle \vdash (a(D)_{s'} \upharpoonright t'(D)_{i,p} \upharpoonright t'(0))}
\]

The pseudocode implementation of \texttt{vmtrigger} is shown in Listings B.4.

**Operation \texttt{vmdiscard}—Discard a timer instance.** On entering operation \texttt{vmdiscard}(\( a, t \)) the kernel adds a reference to timer instance \( j \) to the intentions list \( J \) and returns. Dispatcher invokes
**vmdiscard’** when it encounters a reference to timer instance \( j \) while processing the intentions list \( J \). If the active actor commits, Dispatcher recovers all resources allocated to timer instance \( j \), and sets timer identity \( t \) to **none**. If the active actor aborts message processing, Dispatcher restores timer \( t \) from timer instance \( j \).

**Timer destruction semantics.** The behaviour of operation **vmdiscard** is derived from the semantics for timer destruction described in Sections 3.7, and summarized in transition rule Discard on page 62.

Transition rule Vmdiscard illustrates the outcome of processing a single **vmdiscard** instruction; timer instance \( J_t \) is removed from list of timers \( J \), and a dead timer \( t \), and its owner \( a \) maintain an association.

\[
\begin{align*}
\langle \omega, q, s_c, J \rangle & \vdash (\omega. [\text{vmdiscard}(t) \cdot \text{vmcommit}]_s \mid t_d (D)_{n,p}) \quad \xrightarrow{\tau} \\
[\text{Vmdiscard}] : & \quad \langle \omega, q, s_c, J \rm\ 
\gamma \ (0) \rangle \vdash (\omega. [\text{vmcommit}]_s \mid t_d (D)_{n,p}) \\
& \quad \llbracket K, J - \{J_t\}, \text{tail}(X) \rrbracket \vdash (a. (D)_c \mid t (0))
\end{align*}
\]

The pseudocode implementation of **vmdiscard** is shown in Listings B.5.

### 4.5.3 Message Transmission Operations

In this section we describe the transmission operators provided by the kernel. Under **ACube** semantics all actor communication is managed using a single sequence \( X \). The kernel must manage local messaging (**intranodal**), accommodating communication between actors within a local configuration; and distributed messaging (**internodal**), accommodating actor communication between distinct nodes.

Figure 4.1 illustrates how \( X \) is partitioned to accommodate local and global messaging. Messages enter \( X \) either directly through the intentions list \( J \) into \( X_L \), or from inbound internodal traffic into \( X_I \). Messages leave \( X_L \) either directly from \( X \) when they are dispatched, or from \( X_O \) as out-bound internodal traffic.

In Chapter 2 message delivery in the **Timed Actor model** is described as **not reliable** and **not**
order preserving (see page 29). Figure 4.1 indicates that the kernel utilizes two types of communications media:

- a reliable and order preserving intranodal medium based on the FIFO characteristics of \( X_L \), and

- an unreliable and not order preserving internodal medium based on the connection of \( X_I \) and \( X_O \) to a communications subnet connecting distinct nodes. Messages traversing any communications subnet are subject to loss, corruption and arrival order non-determinism.

**Operation vmsend—Send message.** On entering operation \( \text{vmsend}(a, b, d) \) the kernel performs the following sequence of operations:

1. Allocate a message buffer \( m \) from buffer pool \( B_V \).

2. Initialize message buffer \( m \). This includes setting \( \text{source} \leftarrow a \), \( \text{dest} \leftarrow b \), and \( \text{mid} \leftarrow d \), where \( a \) is the source identity, \( b \) is the destination identity, and \( d \) is the message identity. We employ \( \text{convmid}(\sigma, C) \) to convert message name \( \sigma \) and class name \( C \) to message identity \( d \).

3. Add a reference to message buffer \( m \) to the intentions list \( I \), and return a reference to message buffer \( m \) to the active actor.

Dispatcher invokes \( \text{vmsend} \) when it encounters a reference to message \( m (X&) \) while processing the intentions list \( I \). If the active actor commits; Dispatcher appends the output message \( m \) to \( \text{tail}(X) \). If active actor aborts message processing, Dispatcher returns message buffer \( m \) to buffer pool \( B_V \).

**Transmission semantics—vmsend.** The behaviour of operation \( \text{vmsend} \) is derived from the semantics for message transmission described in Sections 3.5.2, and summarized in transition rule \( \text{Send} \) on page 61.

Transition rule \( \text{Vmsend} \) illustrates the outcome of processing a single \( \text{vmsend} \) instruction; the generated message \( m = \langle a \ b \ f \ v \rangle \) is appended to \( \text{tail}(X) \). The *marshalling* of parameters for message \( m \) is handled by the kernel.
The pseudocode implementation of \texttt{vmsend} is shown in Listings B.5.

\textbf{Operation \texttt{vmforward}—Forward message.} On entering operation \texttt{vmforward}(m,b,d) the kernel performs the following sequence of actions:

1. Allocate a message buffer \(m'\) from buffer pool \(\mathcal{B}_V\).

2. Copy the current input message \(m\) into \(m'\). It is necessary to copy the input buffer \(m\) to the output buffer \(m'\) since operation \texttt{vmdispatch} (see Listing B.2) discards the current input message \(m\) at the commitment point.

3. Initializes message buffer \(m'\). This includes dest \(\leftarrow b\), and mid \(\leftarrow d\), where \(b\) is the destination identity, and \(d\) is the message identity. The identity of the source actor is not modified.

4. Add a reference to message buffer \(m\) (\(X\&\)) to the intentions list \(I\), and return control to active actor.

Operation \texttt{vmforward} is finalized by deferred operation \texttt{vmsend}'.

\textbf{Transmission semantics—\texttt{vmforward}.} The behaviour of operation \texttt{vmforward} is derived from the semantics for message transmission described in Sections 3.5.2, and summarized in transition rule \texttt{Forward} on page 61.

Transition rule \texttt{Vmforward} illustrates the outcome of processing a single \texttt{vmforward} instruction; the input message \(m = \langle \star b \sigma v \rangle\) is appended to \texttt{tail}(\(X\)). The first argument of \texttt{vmforward}, \(m\), is a reference to the input message being processed by actor \(a\). The parameters of message \(m\) may be modified before this instruction is executed.
4.5.4 Commitment Operations

When actors are dispatched their local state is checkpointed. The kernel commitment operators cause the active actor to return the thread-of-control to Dispatcher. When Dispatcher regains control, the kernel immediately processes of the intentions list \( \mathcal{I} \) (see Section 4.4.2).

The effects of \( \text{vmbecomes}(q) \) include; setting the next conceptual state of active actor \( a \) to \( q \), and preserving the modified local state \( s' \) of actor \( a \). The effect of \( \text{vmterminate}() \) is to destroy active actor \( a \). The effects of \( \text{vmabort}() \) include; preserving the current conceptual state of active actor \( a \), and restoring the local state of actor \( a \) from its checkpoint.

Commitment semantics. The behaviour of operations \( \text{vmbecomes} \), \( \text{vmabort} \), and \( \text{vmterminate} \) are derived from the semantics of commitment described in Sections 3.4, and summarized in transition rules \( \text{Becomes} \) on page 58, \( \text{Abort} \) on page 58, and \( \text{Terminate} \) on page 58. The pseudocode implementation of \( \text{vmbecomes} \), \( \text{vmabort} \) and \( \text{vmterminate} \) is shown in Listings B.6.

4.6 Summary

In Chapter 3 we propose a model for an actor system which consists of a configuration \( \langle \mathcal{X}, \mathcal{I}, \mathcal{X} \rangle \), an intentions list \( \mathcal{I} \), and a master-clock \( \mathbb{T} \) (see Figure 3.7). In this chapter we indicate that the model in Figure 3.7 is inadequate to describe a kernel for a timed actor language. The model for the actor system described in this chapter has been implemented and provides a VM to support a timed actor language. Some aspects of this model include:

- The model is restricted to a single-thread of control.
The sequence of pending messages $X$ has been partitioned into three subsequences to more closely model the flow of messages in a global configuration. The model allows us to reference the \textit{locality of actors} or nodes.

The partitioning of $X$ indicates that there are two types of media: one for communication within a local configuration, which is reliable and order preserving; and one for communication within a global configuration, which is unreliable and not order preserving.

The model identifies how resources are used within the VM.

Actors executing in our kernel are described as \textit{asynchronous} with respect to both communication and execution (see Section 4.1). In order to support our message processing model we need to make some assumptions about the execution time of a message handler and the expected delay of a communication. To ensure that execution times for message handlers have low-latency we

- allow actors to process only one message at a time (actors are single-threaded);
- make message processing atomic, allowing the active actor to run to completion (non-preemptive); and
- recommend that high-latency message handlers be divided into a sequence of low-latency message handlers activated by \texttt{self} messages.

We can bound the delay a message experiences by guarding message transmissions with timers.

All actor instances belonging to a class $C$ share the same \textit{actor template} (see Section 4.2). An actor template for class $C$ is an implementation (using VM instructions) of the actor behaviour for actor instances belonging to class $C$. One of the important features of an actor template is that it identifies when actors implicitly raise \textit{exceptions}. In a kernel for a \textit{timed actor language} exceptions are raised transparently to an executing actor, but the handling of exceptions by the kernel can cause an actor to terminate or abort.

Dispatcher is the control element of the kernel which manages all kernel operations and enforces a single-thread of control (see Section 4.3). The polling service implemented by Dispatcher provides the following services:
• Updates the timer list $\beta$, dispatching timeout messages as necessary;

• Checks for inbound messages, moving messages from $\mathcal{R}$ to $\mathcal{X}_I$, dispatching messages as necessary;

• Checks for output messages, moving messages from $\mathcal{X}_O$ to $\mathcal{T}$; and

• Checks local message sequence $\mathcal{X}_L$, dispatching messages as necessary.

• When executing actors reach a commitment point, Dispatcher is responsible for processing the intentions list $\mathcal{J}$.

In Section 4.5 we have extended the timed actor semantics to describe the effect of selected VM instructions on a local configuration at a commitment point. These semantics are incorporated into the implementation for a kernel supporting timed actor semantics presented in Appendix B.2.
Chapter 5

COOL—A Timed Actor Language

In Chapter 2 we introduced a simple timed actor language, STAL, to illustrate the informal semantics of our Timed Actor model. In Chapter 3 we described a timed actor semantics for ACube, an asynchronous actor algebra. This timed actor semantics allows us to model change in an actor system, and to describe how actor behaviour can be realized. In Chapter 4 we described the organization of a virtual machine which accommodates a timed actor language and show how this virtual machine can be derived from the timed actor semantics presented in Chapter 3.

Many actor models have an associated actor language: Atolia is the actor language for the actor model presented in [11], Act is used as an actor language for the actor model presented in [12], and SAL is the actor language for the actor model presented in [1]. In this chapter we define a timed actor language called COOL (Concurrent Object cOordination Language) which incorporates the timed actor semantics of ACube, i.e. the semantics of commitment, asynchronous communication, actor creation, and timers.

COOL is a control language for specifying the interfaces, behaviour, and coordination of actors which implement distributed applications. COOL provides features essential for an effective coordination language such as timer management, the ability to control the degree of concurrency, and the ability to distribute a computation. However, COOL is not a general purpose programming language, and does not provide general data structuring mechanisms. The full language specification of COOL is reported in [30].
COOL shares the following features with other actor programming languages: a **new** operator to create new actor instances, a **send** operator supporting asynchronous message passing, a **becomes** operator to specify *replacement behaviour*, *implicit* message reception, and actor instances with a *completely encapsulated* local state.

Several features distinguish COOL from other actor programming languages: class definitions are structured like *communicating finite state machines*, actors create and manage timers, actors have access to *real time* through a protected instance variable **now**, message processing is *atomic* employing a *local checkpointing mechanism*, message definitions may specify a *drop probability* to simulate *message loss* at run-time, and class definitions can specify a set of *checkable properties* to be monitored at run-time.

**Outline.** The description of COOL is presented using three abstract grammars:

- an abstract grammar of COOL specifications, \( p \in \text{Specification} \), is presented in Section 5.1 (see Figure 5.1),

- an abstract grammar of COOL expressions, \( e \in \text{Exp} \), is presented in Section 5.2 (see Figure 5.2), and

- an abstract grammar of checkable properties, \( \phi \in \text{Properties} \), is presented in Section 6.1 (see Figure 6.2).

Following an approach suggested by [50], each abstract grammar describes the *structure* of the key language features of COOL. In these grammars square brackets [ and ] denote optional sentential forms, hence \( [s_1 \mid s_2 \mid \cdots \mid s_n] \) denotes the selection of 0 or 1 item from \( s_1, s_2, \cdots, s_n \); while \( s_1 \mid s_2 \mid \cdots \mid s_n \) denotes the selection of exactly one item from \( s_1, s_2, \cdots, s_n \). \( s^* \) denotes a sequence which repeats 0 or more times, and \( s^+ \) denotes a sequence which repeats 1 or more times.
5.1 A Syntax and Informal Semantics of COOL Specifications

An abstract syntax of COOL specifications, \( p \in \text{Specification} \), is presented in Figure 5.1. Figure 5.1(a) enumerates the syntactic categories and symbols used in the definition of COOL specifications, and Figure 5.1(b) defines the structure of these syntactic categories.

The elements of a COOL specification are described in Sections 5.1–5.3. Section 5.1 describes the structure of a specification, Section 5.2 describes COOL expressions, and Section 5.3 describes COOL statements. Section 6.1 describes how checkable properties are defined.

Examples to illustrate the syntax of COOL are taken from the specification of alternating bit protocol presented in Listings D.4–D.6 starting on page 231, and the specification of distributed lock protocol presented in Listings D.7–D.10 starting on page 234.

5.1.1 Specifications and Classes

Definition 5.1 (Actor Class) Every actor in COOL is created according to some predefined template of behaviour and interface. This template is called an actor class definition. A specification consists of at least one class definition.

\[
\begin{align*}
\text{Specification} & := \text{classes} \\
\text{classes} & := \text{class}^+ \\
\text{class} & := \text{actor class } C \{ S \text{ blocks } \} \mid \text{actor class } C ;
\end{align*}
\]

where \( C \) denotes the name of the actor class, \( S \) defines the local state of an actor belonging to class \( C \), and \( \text{blocks} \) defines the class body. The COOL class mechanism does not support inheritance. The statement ‘\text{actor class } C ;’ acts as a forward declaration of class \( C \).
p ∈ Specification a COOL specification

class ∈ classes an actor class definition

block ∈ blocks a methods block

function ∈ functions a local function declaration

message ∈ messages a message handler declaration

sdecl ∈ sdecls state variable declaration

vdecl ∈ vdecls method parameter declaration

ldecl ∈ ldecls local variable declaration

guard ∈ guards a message guard

Sdecl ::=

| classes

| class :=

| class := actor class C { S blocks } | actor class C ;

| blocks :=

| blocks := Start : methods block *

| block := q i : methods | Defaults : methods | Satisfies : properties

| methods :=

| methods := functions messages

| statements :=

| statements := control ∪ replace

| functions :=

| functions := function *

| function :=

| function := fun d m(V) \{ D ; c_f \}

| c_f :=

| c_f := c_f ; c_f ; | x = e | m(e,e',⋯) | if (e) \{ c_f \} else \{ c_f \} | discard t | return e

| messages :=

| messages := message *

| message :=

| message := msg m(V) G [ dprob e ] \{ D ; c^e ; r \}

| c :=

| c := c ; c ; | x = e | m(e,e',⋯) | if (e) \{ c^e ; r \} else \{ c^e ; r \}

| | m(e',e''⋯) ⇒ e ⇒ e | discard t

| r :=

| r := becomes q | commit | abort | terminate

| guards :=

| guards := guard *

| guard :=

| guard := [ from x | if (e) ]

| d :=

| d := void | bool | int | real | string | actor | Timer | time | C

| sdecls :=

| sdecls := sdecl *

| sdecl :=

| sdecl := d x,x',⋯ ; | [ const ] d x = e ;

| ldecls :=

| ldecls := ldecl *

| ldecl :=

| ldecl := d x,x',⋯ ; | d x = e ;

| vdecls :=

| vdecls := vdecl *

| vdecl :=

| vdecl := d x | d x,

(a) Syntactic Categories

Figure 5.1: An abstract syntax of COOL specifications, p ∈ Specification.

For example, the specification of a distributed lock protocol (see Listings D.7–D.9) consists of three class definitions.
actor class DLock {
    Timer t;
    ...
}
actor class Locker {
    int n1, n2, n;
    ...
}
actor class User {
    Dlock d;
    ...
}

A COOL application is a collection of actor instances. Every actor is instantiated with a single thread of control, a globally unique identity, a private local state (possibly including a set of timers), a private function interface, and public message interface. Message passing is the only mechanism of interaction between actors; actors react only to the reception of messages.

5.1.2 Method Blocks

The body of an actor class definition is divided into a finite number of method blocks and an optional properties block. Each method block corresponds to a conceptual state, $q_i \in Q$, and contains a finite number of function and message handler declarations. This block structure models a communicating finite state machine (see Appendix A.2 on page 204).

$$
\text{blocks} ::= \text{Start} : \text{methods} \quad \text{block}^* \\
\text{block} ::= q_i : \text{methods} \mid \text{Defaults} : \text{methods} \mid \text{Satisfies} : \text{properties} \\
\text{methods} ::= \text{functions} \quad \text{messages}
$$

Message declarations describe a public communication interface to actor instances belonging to class $C$, while function declarations describe a private method interface available to actor instances belonging to class $C$.

Every actor class definition must include a method block labelled $\text{Start} \in Q$ which defines the initial behaviour of actor instances belonging to class $C$. A method block labelled $\text{Defaults} \notin Q$ defines messages accessible in all conceptual states.
An actor class specification may contain a *properties block* labelled **Satisfies** $\not\in Q$ which defines a set of *checkable properties* to monitor at run-time. Checkable properties are described in Chapter 6.

```
actor class DLock {
    ... 
    Start:
    msg init(int x) { ... }
    Srv:
    msg lock(actor u) { ... }
    msg unlock() { ... }
    msg noquantum() from t if (h != none) { ... }
    Defaults:
    msg stop() from owner { ... }
    Satisfies:
    // A checkable property; the binary lock invariant
    $\phi_1 \triangleq 0 \leq \#!\text{locked} - \#!\text{unlocked} \leq 1$
}
```

### 5.1.3 Type Declarations

A data type determines the set of values that variables of that type may assume and the operators that are applicable [51]. A type declaration is used to associate an identifier with the type. COOL provides no explicit mechanism for defining *structured* data types; however, message passing can be seen as an implicit mechanism for structuring data. COOL supports the following *basic* data types:

- **bool**—a boolean value (default value is `false`),
- **int**—an integer value (default value is 0),
- **real**—a floating-point value (default value is 0.0),
- **string**—a string constant (default value is ""),
- **time**—a master-clock ($T$) value (default value is 0),
- **actor**—the identity of an actor (default value is `none`), and
- **Timer**—the identity of a timer (default value is `none`).
Every actor class \( C \) also defines a new type \( C \), which is the set of all instances of class \( C \). Any actor declared to be of type \texttt{actor} is called an \textit{anonymous} actor; otherwise, it is called a \textit{regular} actor.

```cpp
actor class DLock {
    actor h, k; // anonymous actors
    ...
}
actor class User {
    DLock d; // a regular actor
    ...
}
```

Type \texttt{void} is used to describe functions with no parameters or return type.

### 5.1.4 Variable Declarations

In \texttt{COOL}, variables are declared at three scoping levels. A variable \( x \in sdecls \) exists at \textit{state scope}, a variable \( x \in ldecls \) exists at \textit{local scope} (within the \textit{body} of a function or message handler), and a variable \( x \in vdecls \) exists at \textit{parameter scope} (within the \textit{header} of a function or message handler).

State and local variable declarations can include initializers.

\[
sdecls ::= sdecl^* \\
sdecl ::= d x, x', \cdots; | [\textbf{const}] d x = e; \\
ldecls ::= ldecl^* \\
ldecl ::= d x, x', \cdots; | d x = e; \\
vdecls ::= vdecl^* \\
vdecl ::= d x | d x,
\]

As with many imperative languages, variables declared at an inner scoping level can hide variable declarations in an outer scoping level; for example, a variable declared at local scope (innermost level) will hide a variable with the same name declared at state scope (outermost level) or parameter scope.

```cpp
actor class P {
    int y = 5;
    msg m(int x, actor a) {
        int y; // Declaration of local variable y hides
    }
}
```
\[ y = x \times 5; \quad \text{// declaration of variable } y \text{ in state block.} \]

Every actor has access to the protected instance variables listed in Table 5.1 (also see Section 3.8.1). These variables cannot be hidden, redefined, or directly modified by an actor.

Table 5.1: Protected instance variables.

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Scope</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>self</td>
<td>state</td>
<td>C</td>
<td>The identity of an actor.</td>
</tr>
<tr>
<td>owner</td>
<td>state</td>
<td>actor</td>
<td>The identity of the owner of an actor.</td>
</tr>
<tr>
<td>current</td>
<td>state</td>
<td>int</td>
<td>The current conceptual state of an actor.</td>
</tr>
<tr>
<td>now</td>
<td>state</td>
<td>time</td>
<td>The activation time (T) of an actor.</td>
</tr>
</tbody>
</table>

**Constant Declarations.** A constant declaration associates a variable with a read-only value. In COOL constants are introduced with the `const` keyword. Constants can only be declared at state scope and are evaluated at run-time immediately after an actor is created. This means that constant declarations are not restricted to using literal constants.

```c
const real M_PI = 3.14159265358979323846;
const int x = fixpoint( 40, 0.25 );
```

```c
fun int fixpoint( int x, real y) { ··· }
```

**5.1.5 Methods**

**5.1.5.1 Functions**

Function declarations describe the private interface of an actor class,

\[
\begin{align*}
\text{functions} & ::= \text{ function}^* \\
\text{function} & ::= \text{ fun } d \ m(V) \ \{ D ; c_f^* \} \\
\text{c_f} & ::= \ c_f ; \ c_f ; \ x = e \ | \ m(e,e',\cdots) \ | \ \text{if} (e) \ \{ c_f^* \} \ \text{else} \ \{ c_f^* \} \ | \ \text{discard} \ t \ | \ \text{return} \ e
\end{align*}
\]

where \( d \) is a function return type, \( m \) is a function name, \( V \) declares the formal parameters of a function, \( D \) declares the local variables of a function, and \( c_f^* \) is a sequence of control statements
executed when function \( m \) is invoked. Function \( m \) may be defined anywhere in the class body and is accessible in any conceptual state. Functions cannot be overloaded or redefined.

All instances of an actor class share the same function interface. Function declarations consist of a *heading* and a *function body*. The heading specifies the result type, the function name and the *formal parameters*. The body contains declarations of local variables and COOL statements.

Functions can be used as either *proper functions* or *procedures*. Proper functions are activated by a reference to a function in an expression, and yield a result that is an operand in the expression [51]. Procedures are activated as statements. A proper function is distinguished in a declaration by the presence of a non-void result type preceding the method name. Its body must contain a *return* statement that defines the result of the function activation. The return type of a procedure is *void*.

```plaintext
actor class Sender {

    .../

    fun int inc(int x, int y) { return (x + 1) % (y + 1); }
    fun int dec(int x, int y) { return (x + y) % (y + 1); }

    Start:

    .../
```

Local variable declarations are constructed in exactly the same manner as state variable declarations except that the *const* modifier is not allowed at local scope. Functions can be defined with formal parameters and return values of any type. Functions may access the state variables.

Functions do not have access to the general set of control and commitment statements described by the abstract grammar in Figure 5.1; in particular, functions cannot generate messages or utilize the commitment statements.

### 5.1.5.2 Message Handlers

Message handler declarations describe the public interface of an actor,

```plaintext
messages ::= message^+  
message ::= msg m(V) G [dprob e] \{D ; c^* ; r\}  
c ::= c ; c | x = e | m(e, e', ..., m(e'', e''' ...)) | if (e) \{c^* ; r\} else \{c^* ; r\}  
r ::= becomes q | commit | abort | terminate
```
where \( m \) is the name of a message handler, \( V \) declares the formal parameters of a message handler, \( G \) defines an input guard, \( \text{dprob} \) establishes a drop probability, \( D \) declares the local variables of a message handler, \( e^* \) is a sequence of control statements executed when message \( m \) is received, and \( r \) defines the commitment behaviour of an actor. Message \( m \) may be defined in more than one method block but cannot be overloaded.

Local variable declarations are constructed in exactly the same manner as state variable declarations except that the \texttt{const} modifier is not allowed at local scope. Message handlers can be defined with formal parameters of any type, except type \texttt{Timer}. Message handlers may access the state variables.

```plaintext
actor class DLock {
    Timer t;
    actor h, k;
    int quanta;

    Srv:
    msg lock(actor u) {
        if (h == none) {
            t = trigger noquantum() on quanta;
            h = *;
            locked() ⇒ h;
        }
    }
    msg unlock() {
        if (h == *) {
            discard t;
            unlocked() ⇒ h;
            k = h; h = none;
        }
    }
}
```

Message handlers have access to the general set of control and commitment statements described by the abstract grammar in Figure 5.1. These statements are divided into four groups.

1. Actor management operations which are described in Section 5.2.2 include operator \texttt{new} and statement \texttt{terminate}. Actor \( a \) creates actor instance \( b \) of class \( C \) with expression \( b = \text{new} C \). An actor \( b \) can self-terminate during any of its activation using statement \texttt{terminate}. 

2. Timer management operations which are described in Section 5.2.3 include operation \texttt{trigger} and statement \texttt{discard}. An actor \( a \) creates an oneshot timer \( t \) with duration \( n \) using expression \( t = \texttt{trigger}(m(e)) \text{ on } n \), or it creates a periodic timer \( t \) with period \( n \) with expression \( t = \texttt{trigger}(m(e)) \text{ every } n \). When timer \( t \) expires it sends message \( m(e) \) to its owner. The owner of timer \( t \) destroys \( t \) with statement the \texttt{discard} \( t \).

3. Message transmission statements which are described in Section 5.3.2 include the send and forward statements. An actor \( a \) sends message \( m(e) \) to actor \( b \) using statement \( m(e) \Rightarrow b \). When actor \( a \) is processing message \( m(v) \), a copy of message \( m(v) \) can be transmitted to actor \( b \) using statement \( \Rightarrow b \).

4. Commitment statements which are described in Section 5.3.3 provide a \textit{local checkpointing} mechanism. \textit{COOL} provides statements \texttt{commit, becomes q, terminate} and \texttt{abort} for this purpose.

**Input guards.** In \textit{ACube} receive operations \texttt{rcv(a,m(v))} message \( m \) will be accepted by the target actor if the identity of the source actor is equal to \( a \). In this statement \( a \) acts as an input guard for message \( m \). Receive operation \texttt{rcv(\*,m(v))} accepts message \( m \) from any actor. In \textit{COOL} this type of input guard is introduced with the \texttt{from} clause. The absence of a \texttt{from} clause is equivalent to a receive operation of the form \texttt{rcv(\*,m(v))}.

\begin{verbatim}
actor class Receiver {
    Sender s;
    Start:
        msg init(Sender x) { } // rcv(*,init(x))
    Rx:
        msg f(int seq, int x) from s { } // rcv(s,f(seq,x))
}
\end{verbatim}

A more general input guard is provided by the \texttt{if} clause.

\begin{verbatim}
actor class Sender {
    Tx:
        msg g(int seq) if (r == *) { }
}
\end{verbatim}
**Drop probability.** To validate an implementation it is often necessary to examine the behaviour of actors in a lossy environment. To simulate message loss, a *drop probability* can be associated with any message definition in any conceptual state. We assign the drop probability $e$ to a message handler using a `dprob` clause,

```latex
msg f(int seq, int x) dprob e {
... 
}
```

where $e$ evaluates to a real-valued constant, $0.0 \leq e \leq 1.0$.

During execution, before an actor receives a dropable message $f()$, the run-time system draws a sample $y$ from the *uniform distribution* $U(0,1)$. If $y > e$ message $f()$ will be received; otherwise when $y \leq e$ message $f()$ is dropped and the active actor *aborts* preserving its local state.

We examined two general approaches before deciding on the syntax of the `dprob` clause:

1. specify the drop probability as part of a transmission statement, or

   ```latex
   f(0,0) dprob 0.01 => a;
   ```

2. specify the drop probability at the receiver as part of the *message header*.

   ```latex
   S_1:
   msg f(int seq, int x) dprob 0.01 { ... }
   S_2:
   msg f(int seq, int x) dprob 0.15 { ... }
   ```

Both approaches are easily implemented, but the second approach is preferred since as the example shows, any message may have multiple definitions in the target class. This allows us to remove certain conceptual states from our testing or to apply differential drop probabilities in selected conceptual states. The first approach does not allow for this type of control.

**Input messages.** The set of messages that an actor may receive is called its *input set*, which is specified by the message interface of its actor class. If $a$ is an instance of an actor class $C$, then $\Sigma_a = \Sigma_C$ and $\Sigma_C$ is the set of input messages for actor $a$. Definition 5.2 introduces the concept of a *legal message* $m \in \Sigma_C$. 
Definition 5.2 (Legal Messages) The sending of a message \( m(v_1, \ldots, v_n) \) to an actor \( a \) is legal if \( m(d_1, \ldots, d_n) \) is in \( \Sigma_a \) and for each \( i = 1 \cdots n \),

- \( d_i \) is not \texttt{Timer} (\texttt{Timer} identities cannot be transmitted),
- \( d_i \) is \texttt{bool, int, real, string, time} and \( v_i \) is of type \( d_i \),
- \( d_i \) is a regular actor class \( C \), and \( v_i \) is of type \( C \), or
- \( d_i \) is \texttt{actor}, and \( v_i \) is a regular or anonymous actor.

Union sigma. If a specification \( p \) consists of \( n \) actor class definitions, \( \Sigma_{\text{actor}} \) (pronounced “union sigma”) represents the set of messages any actor instance in \( p \) may send.

\[
\Sigma_{\text{actor}} = \bigcup_{i=1}^{n} \Sigma_C \tag{5.1}
\]

For the distributed lock protocol in Listings D.7–D.9, \( \Sigma_{\text{actor}} \) is the union of the input sets for classes \( \text{DLock}, \text{Locker}, \text{and User} \).

\[
\begin{align*}
\Sigma_{\text{DLock}} &= \{ \text{init(int), lock(actor), unlock(), noquantum(), stop} \} \\
\Sigma_{\text{Locker}} &= \{ \text{init(int,int), lock(actor,actor), locked(actor), locked_r(actor),} \\
& \quad \text{unlock(actor), unlocked(), unlocked_r(), reject(), noresponse(int), stop} \} \\
\Sigma_{\text{User}} &= \{ \text{init(DLock), nextAttempt(), locked(actor), complete(), expired(),} \\
& \quad \text{unlocked(), stop} \} \\
\Sigma_{\text{actor}} &= \{ \text{init(DLock), init(int), init(int,int), lock(actor), locked(actor),} \\
& \quad \text{locked_r(actor), nextAttempt(), unlock(), unlock(actor), unlocked(), unlocked_r(),} \\
& \quad \text{reject(), noresponse(int), noquantum(), stop(), complete(), expired} \}
\end{align*}
\]

Predefined messages. Every actor must accept an \texttt{init()} and a \texttt{stop()} message from its owner. The \texttt{init()} message (declared only in method block \texttt{Start}) allows an owner to initialize the local state of a child actor. When an actor receives a \texttt{stop()} message (a redefinable message typically declared in method block \texttt{Defaults}) from its owner, it must terminate.

\begin{verbatim}
actor class DLock {
    Start:
        \texttt{init(int), lock(actor), unlock(), noquantum(), stop} 
    }
\end{verbatim}
msg init(int x) from owner { ⋯ ; becomes Srv ; }
Defaults:
msg stop() from owner { ⋯ ; terminate ; }
}

5.2 A Syntax and Informal Semantics of COOL Expressions

Expressions are constructs denoting rules of computation wherein constants and the current value of variables are combined to derive new values. This is achieved by the application of predefined operators and user-defined functions.

The syntax and semantics of COOL expressions are defined to be as close to its implementation language as possible, but still have enough expressiveness to allow the definition of ad hoc computations. The following categories of COOL expressions require comment:

- type tests and type guards (see Section 5.2.1),
- operations for actor management (see Section 5.2.2),
- operations for timer management (see Section 5.2.3), and
- operations for accessing the local master-clock (see Section 5.2.3).

An abstract syntax of COOL expressions, $e \in Exp$, is presented in Figure 5.2. Figure 5.2(a) enumerates the syntactic categories and symbols used in the definition of COOL expressions, and Figure 5.2(b) defines the structure of these syntactic categories. COOL supports the arithmetic, logical and relational operators listed in Table 5.2.

5.2.1 Type Casts, Type Tests, and Type Guards

If $x$ is an anonymous actor, a type cast $C(x)$, assumes the run-time type of $x$ to be $C$. COOL type casts can only be applied to actor variables.
An anonymous actor is *polymorphic*, that is, its run-time type may change from time to time.
The `is` operator is used to test the run-time type of an anonymous actor. The result of applying the type test, \( x \text{ is } C \), to anonymous actor \( x \) is **true** if the run-time type of \( x \) is class \( C \), otherwise the result is **false**.

A **type guard** takes the form \((C)x\) and can be applied to any anonymous actor. This operation returns \( x \) if \( x \) belongs to class \( C \), otherwise it returns **none**.

```plaintext
actor class R;
actor class P;
actor x;
P y = (P)x;       // a type guard
R z = R(x);      // a type cast
if (z is R) { ... } // a type test
```

### 5.2.2 Actor Management Operations

The **new** operator is used to create an instance of an actor class. The argument to the **new** operator is a class name. The result of this operation may be assigned to actor or class variables, subject to the **assignment rules** discussed in Definition 5.5 on page 119. All new actors begin their lifetime in conceptual state **Start**. The actor executing the **new** operator **owns** the newly created actor instance.

A run-time system for COOL must guarantee that

- state variables declared with initializers are evaluated before the initial activation of a new actor, and

- state variables declared without initializers will be assigned the following initial values before the initial activation of a new actor: **bool** variables are initialized to **false**, **int** and **time** variables are initialized to 0, **real** variables are initialized to 0.0, **string** variables are initialized to "", and **actor** (anonymous and regular) and **Timer** variables are initialized to **none**.

**Actor initialization.** When an actor is created using the **new** operator, it is ready to receive messages. To initialize the internal state of an actor, we send it an **init()** message. The creation of an actor and its initialization are decoupled. Initialization must often be deferred until all appropriate
state information is available. For example, if an actor p must know the identity of an actor b at initialization we write

```java
Buffer b = new Buffer;
Producer p = new Producer;
Consumer c = new Consumer;
init(p, c) ⇒ b;
init(b) ⇒ p;
init(b) ⇒ c;
```

**Actor termination.** An actor is permanently removed from a system by executing a `terminate` statement (explicit) or by accepting a `stop()` message from its owner (implicit). There is no support for automatic garbage collection of actors in COOL. The resources associated with an actor can only be reclaimed by the run-time system when an actor terminates.

### 5.2.3 Timing and Timer Management Operations

**Definition 5.3 (Timers)** Timers are an integral part of COOL. An actor may create as many timers as it desires. Each timer is unique. When a timer expires, it sends a user-defined timeout message to its owner. If the timer was created as a oneshot timer it terminates automatically. If the timer was created as a periodic timer it is automatically restarted with a predeclared period.

A predefined type `Timer` together with the `trigger` operator and `discard` statement form the basis of the timer mechanism in COOL. `trigger` expressions are used to create and activate `Timer` objects. `trigger` expressions are only legal in assignments to `Timer` variables, and `Timer` variables can only be declared at state scope. Timers are owned by the actor which creates them.

The `trigger` operator specifies: the name of a timeout handler, the type of timer to create, and, the duration of the timer. COOL supports oneshot (on) and periodic (every) timers. The duration clause indicates the interval of a oneshot timer or the period of a periodic timer and can include one of the following time grain operators: `sec` (second), `msec` (millisecond) or `usec` (microsecond). `msec` is the default grain.
When a timer expires, the run-time system delivers a timeout message to the owner of the timer. Timeout messages have exactly the same structure as messages exchanged by actors. The discard statement is used to halt and remove an active timer. A trigger operation on an already active timer (referred to as retriggering) implicitly discards the timer. That is, the following sequence

```
Timer t;
t = trigger delay() on 30 msec;
t = trigger delay() on 40 msec;
```

is equivalent to

```
Timer t;
t = trigger delay() on 30 msec;
discard t; // implicit discard
t = trigger delay() on 40 msec;
```

**Definition 5.4 (Timing and the Local Master-clock)** Every actor has a read-only instance variable called `now`, which is assigned the current value of the local master-clock ($T$) each time the actor is activated (dispatched). The `now` instance variable is of type `time`. The processing of a message by an actor takes zero time, so the value of `now` does not change during the current activation.

COOL provides a `time` type and the set of operations listed in Table 5.2 to manipulate `time` values. Arithmetic operations involving time variables may also include the grain declarations; `sec` (second), `msec` (millisecond), or `usec` (microsecond). `usec` is the default grain for arithmetic expressions.

The following example demonstrates how to estimate the actual duration of timer $t$ by subtracting the activation time of message `init()` from the activation time of message `measure()`.

```
actor class MeasureTrigger {
    time mark;
    Timer t;
    msg init(int x) {
        mark = now;
        t = trigger measure() on x usec;
    }
    msg measure() {
        int elapsed = (now - mark) usec;
    }
}
Table 5.2: Predefined COOL operators.

<table>
<thead>
<tr>
<th>Category</th>
<th>op</th>
<th>d → d'</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Boolean Operators (bop)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical Operators</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>bool × bool → bool</td>
</tr>
<tr>
<td></td>
<td>!</td>
<td>bool → bool</td>
</tr>
<tr>
<td>Equality Operators</td>
<td>==</td>
<td>d × d' → bool, d = d'</td>
</tr>
<tr>
<td></td>
<td>!=</td>
<td>d × d' → bool, d = d'</td>
</tr>
<tr>
<td>Ordering Operators</td>
<td>&lt;</td>
<td>d × d → bool, d = d' for d ∈ {int, real, time}</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>d × d → bool, d = d' for d ∈ {int, real, time}</td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
<td>d × d → bool, d = d' for d ∈ {int, real, time}</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>d × d → bool, d = d' for d ∈ {int, real, time}</td>
</tr>
<tr>
<td><strong>Integer Operators (iop)</strong></td>
<td>+</td>
<td>int × int → int</td>
</tr>
<tr>
<td></td>
<td>−</td>
<td>int × int → int</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>int × int → int</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>int × int → int</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>int × int → int</td>
</tr>
<tr>
<td>Increment Operators</td>
<td>++</td>
<td>int → int</td>
</tr>
<tr>
<td></td>
<td>−−</td>
<td>int → int</td>
</tr>
<tr>
<td>Shift Operators</td>
<td>&lt;&lt;</td>
<td>int × int → int</td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;</td>
<td>int × int → int</td>
</tr>
<tr>
<td>Bitwise Operators</td>
<td>&amp;</td>
<td>int × int → int</td>
</tr>
<tr>
<td></td>
<td>^</td>
<td>int × int → int</td>
</tr>
<tr>
<td></td>
<td>~</td>
<td>int → int</td>
</tr>
<tr>
<td><strong>Real Operators (rop)</strong></td>
<td>+</td>
<td>real × real → real</td>
</tr>
<tr>
<td></td>
<td>−</td>
<td>real × real → real</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>real × real → real</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>real × real → real</td>
</tr>
<tr>
<td><strong>Time Operators (zop)</strong></td>
<td>+</td>
<td>time × int → time</td>
</tr>
<tr>
<td></td>
<td>−</td>
<td>time × time → int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>time × int → time</td>
</tr>
</tbody>
</table>
5.3 Statements

Every dynamic ACube operator has a corresponding statement in COOL. This correspondence is shown in Table 5.3. The following categories of COOL statements require comment:

- the assignment statement (see Section 5.3.1),
- statements for message transmission (see Section 5.3.2), and
- commitment statements (see Section 5.3.3).

5.3.1 Assignment Statement

Assignment serves to replace the current value of a variable by a new value specified by an expression [51]. The type of the expression must match the type of the variable, or it must be assignment compatible. Rules for assignment compatibility are introduced in Definition 5.2 on page 112. There are special rules governing assignment to actor and class variables.

**Definition 5.5 (Actor Assignability)** There are two distinct categories of actors: anonymous and regular. An actor a is assignable to another actor b if

- both a and b are regular actors and belong to the same actor class,
- both a and b are anonymous actors, or
- b is an anonymous actor and a is a regular actor.

The constant `none` may be assigned to any actor, class or Timer variable.

The ACube operator `{x ← e}` is equivalent to the COOL assignment statement `x = e`. In COOL actors access an additional scoping level (local scope). There is no `scope qualifier` in COOL.
Table 5.3: A comparison of ACube operators and COOL statements.

<table>
<thead>
<tr>
<th>Description</th>
<th>ACube Operator</th>
<th>VM instruction(^1)</th>
<th>COOL Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Statements Common to ACube and COOL</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence operator</td>
<td>(c \cdot c</td>
<td>c \cdot r)</td>
<td>(c ; c ;</td>
</tr>
<tr>
<td>Null operator</td>
<td>skip</td>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>Assignment operator</td>
<td>({x \leftarrow e})</td>
<td>(x = e)</td>
<td></td>
</tr>
<tr>
<td>Selection operator</td>
<td>if ((e)) {(c^<em>{r})} else {(c^</em>{r})}</td>
<td>if ((e)) {(c^<em>{r})} else {(c^</em>{r})}</td>
<td>if ((e)) {(c^<em>{r})} else {(c^</em>{r})}</td>
</tr>
<tr>
<td>Create actor</td>
<td>new((a, C))</td>
<td>(a = \text{vmnew}(\text{self}, C))</td>
<td>(a = \text{new} C)</td>
</tr>
<tr>
<td>Discard actor</td>
<td>terminate</td>
<td>\text{vmterminate}()</td>
<td>\text{terminate}</td>
</tr>
<tr>
<td>Create oneshot timer</td>
<td>trigger((t, m(e), i, \text{oneshot}))</td>
<td>(t, b = \text{vmtrigger}(\text{self}, t, \text{convmid}(m, C_d), i, \text{oneshot}))</td>
<td>(t = \text{trigger} m(e) \text{ on } i)</td>
</tr>
<tr>
<td>Create periodic timer</td>
<td>trigger((t, m(e), i, \text{periodic}))</td>
<td>(t, b = \text{vmtrigger}(\text{self}, t, \text{convmid}(m, C_d), i, \text{periodic}))</td>
<td>(t = \text{trigger} m(e) \text{ every } i)</td>
</tr>
<tr>
<td>Discard timer</td>
<td>discard((i))</td>
<td>\text{vmdiscard}(\text{self}, t)</td>
<td>\text{discard} (i)</td>
</tr>
<tr>
<td>Send message</td>
<td>send((a, m(e)))</td>
<td>(b = \text{vmsend}(\text{self}, a, \text{convmid}(m, C_d)))</td>
<td>(m(e) \Rightarrow a)</td>
</tr>
<tr>
<td>Forward message</td>
<td>forward((a))</td>
<td>\text{vmforward}(b, a, \text{convmid}(m, C_d))</td>
<td>(\Rightarrow a)</td>
</tr>
<tr>
<td>Commit, change conceptual state</td>
<td>becomes((q))</td>
<td>\text{vmbecomes}((q))</td>
<td>\text{becomes} (q)</td>
</tr>
<tr>
<td>Abort processing</td>
<td>abort</td>
<td>\text{vmbecomes}()</td>
<td>\text{abort}</td>
</tr>
<tr>
<td><strong>COOL Extensions</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function invocation</td>
<td>(f(e, e', \cdots))</td>
<td>(f(e, e', \cdots))</td>
<td></td>
</tr>
<tr>
<td>Return from function</td>
<td>return (e, e', \cdots)</td>
<td>\text{return} (e)</td>
<td></td>
</tr>
<tr>
<td>Commit, preserve conceptual state</td>
<td>becomes((\text{current}))</td>
<td>\text{vmbecomes}((\text{current}))</td>
<td>\text{commit}</td>
</tr>
</tbody>
</table>

\(^1\)In the VM instructions \(b\) is a reference to a message buffer and \(C_d\) is the class name of the destination actor.
5.3.2 Message Transmission Statements

In COOL the ACube send operator send\( (a, m(e)) \) is written as \( m(e) \Rightarrow a \). Forwarding an activation message to actor \( a \) is denoted by \( \Rightarrow a \).

**Definition 5.6 (Messages)** Actors communicate with each other via non-blocking asynchronous messages. A message has a functional syntax. The expression \( m(x, y, z) \) denotes a message with name \( m \) and arguments \( x, y, \) and \( z \).

To send a message \( m(x, y, z) \) to an actor \( a \), we write

\[
m(x, y, z) \Rightarrow a
\]

When an actor sends a message to itself, we write \( m(x, y, z) \Rightarrow \text{self} \), where \( \text{self} \) is a special keyword which denotes the active actor. If actor \( a \) receives a message \( g(u, v) \) from actor \( b \) and wishes to reply with message \( f(x, y, z) \), we write \( f(x, y, z) \Rightarrow * \), where \( * \) denotes the sender of the current message, in this case actor \( b \). The notation \( \Rightarrow r \) means forward the current message to actor \( r \) as if the message were originally destined to actor \( r \). For example

```java
actor class P {
    int x, y, z;
    actor r;
    ...
    msg h( int u ) {
        h( u+1 ) \Rightarrow \text{self};
    }
    msg g( int u, int v ) {
        f( x+u, y+v, z ) \Rightarrow * ;
    }
    msg f( int u, int v ) {
        \Rightarrow r;
    }
}
```

Every transmitted message is typed and has exactly one receiver. Intrinsically, each message contains the identity of the sending actor, the identity of the receiving actor, the message name and arity, and a communication (message contents). Sending a message to a nonexistent actor or \( \text{none} \) is a null operation. Not all messages sent to an actor are legal (see Definition 5.2 on page 112). One
function of a compiler for the COOL language is to minimize the number of illegal messages sent to a receiver. Every transmitted message is checked by the compiler for acceptance. The following paragraphs indicate some of the decisions made by a COOL compiler while checking transmissions for acceptance.

Transmission to regular actors. Send and forward statements where the destination is a regular actor can be unambiguously resolved at compile-time using the type information associated with the destination variable. In the following example, an instance of class R can always send message f(int,int) to an instance of class P.

```java
actor class P {
    msg init() { ... }
    msg f(int x, int y){ ... }
}
actor class R {
    P p = new P;
    msg init() { ⇒ p; f(0,1) ⇒ p; }
}
```

Transmission to anonymous actors. When messages are sent to anonymous actors a number of strategies are employed to insure that a receiver only receives legal messages. This process is somewhat exacerbated by the rules of actor assignability (Definition 5.5 on page 119), which allows actor message parameters to accept both actor and class actual parameters.

Type casts. Type casts, P (p), are employed in situations where we are certain of the class associated with the receiving actor.

```java
actor class R {
    actor p = new P;
    msg init() { ⇒ P(p) ; f(0,1) ⇒ P(p); }
}
```

Casting is resolved at compile-time. The associated send or forward operation may fail if the cast operator is applied to an anonymous actor that does not belong to the specified class.
**Type guards.** *Type guards, \((P)\)\(p\), are employed in situations where we are uncertain of the class associated with a destination actor.*

```java
type class R {
  msg f ( ) { ⇒ (P) * ; g(0) ⇒ (P) * ; }
}
```

With *type guards*, a compiler is able to check the message signature using the class information associated with the destination actor. However, the validity of the message can only be guaranteed by applying a *type test* on the destination actor variable at run-time (see Section 5.2.1). If the *type test* returns *true* the associated send (forward) statement will be executed, otherwise the message is not sent and execution proceeds immediately after the send (forward) statement. In a COOL specification a *type guard* can always be replaced by a *type test* followed by a *type cast*.

```java
type class R {
  msg f ( )
  {
    if ( * is P) {
      ⇒ P ( * );
      g(0) ⇒ P ( * );
    }
  }
}
```

**Using run-time type information.** In situations where class information is not immediately available, a compiler searches \(\Sigma_{\text{actor}}\) to identify classes with message signatures matching the current transmission statement. For a send statement the compiler infers a message signature from the types of the actual message arguments. For a forward statement the compiler uses the signature of the message handler currently being compiled. In both cases the message signature must be an element of \(\Sigma_{\text{actor}}\) for the compile-time check to succeed.

If a message signature is found, \(f(d_1, d_2, \cdots) ∈ \Sigma_{\text{actor}}\), the compiler generates code to validate the transmission operation at run-time. This is accomplished by searching a run-time version of \(\Sigma_{\text{actor}}\) for run-time type information [see 14, page 108]. In the following example, the signature of statement \(f(4, \text{true}) ⇒ *\) is \(f(\text{int, bool})\) which matches the declaration of message \(f\) in classes \(P\) and
R. For this statement, the compiler will generate code which ensures that message \( f \) reaches the correct destination.

\[
\text{actor class } P \{ \\
\quad \text{msg } f(\text{int } x, \text{ bool } y) \{ \cdots \}
\}
\]

\[
\text{actor class } R \{ \\
\quad \text{msg } f(\text{int } x, \text{ bool } y) \{ \cdots \}
\}
\]

\[
\text{actor class } C \{ \\
\quad \text{msg } g() \{ f(4, \text{true}) \Rightarrow * ; \}
\}
\]

### 5.3.3 Commitment Statements

In COOL message processing is \textit{atomic}; processing either completes successfully or it is abandoned. When an actor accepts a message it is required to finalize its processing by executing one of the following commitment operations:

- **commit**—when it successfully processes the current input message and wishes to preserve its current conceptual state,
- **becomes } q —when it successfully processes the current input message and wishes to transition to conceptual state } q,
- **terminate**—when it successfully processes the current input message and self-terminates, or
- **abort**—when it abandons processing the current input message. When an actor aborts, all changes made while processing the current input message are undone, i.e.,
  
  - the local state of the active actor is restored from its most recent checkpoint,
  - all actors created are removed,
  - all timers created are discarded,
  - all timers discarded are restored, and
  - all generated messages are discarded.
If a commitment statement is not specified in a message handler the default action is **commit**. The actions **abort**, **commit**, **becomes**, and **terminate** may appear anywhere within a message handler definition, and are taken immediately. Function definitions cannot include commitment statements.

### 5.3.4 Run-time Exceptions

COOL provides no mechanism for declaring, explicitly raising, or handling run-time exceptions. In a COOL application exceptions are raised and handled *transparently* by the run-time system (see Section 4.4.3). Exceptions can arise when input messages are accepted, when messages are transmitted, when resources become exhausted, and when the *current conceptual state* of an actor becomes undefined.

All nonresource related exceptions map input and output messages into dropped (lost) messages. The possibility of dropped messages must always be anticipated in the design of a COOL application.

### 5.4 Summary

COOL is an object coordination language used to implement distributed applications. The purpose of COOL is to provide a high-level description language for specifying the interfaces, behaviour and coordination of active objects (actors and timers).

In COOL, every actor is created according to some predefined template of behaviour and interface. This template is called an **actor class definition**. Every actor instance has a private local state and its own thread of control. The message interface of an actor class is public, defining what messages may be accepted by an actor belonging to the actor class. Actors communicate via asynchronous messages. The sender is not required to wait until the message has been transmitted or wait for a response. A message sent to an actor, which is not in the *message set* of the recipient, is ignored.

An actor is **ready** the instant it is created (**new** is used to create actors). It remains **ready** until it receives a message, at which time it becomes **active**. Once **active** an actor cannot be preempted
until it executes one of the commitment statements (abort, commit, terminate or becomes).

Actors behave like finite state machines, where messages serve as the input to the state machine. Taking a transition in the state machine corresponds to processing an input message. The input set of an actor is the input alphabet of the state machine, and specifying the behaviour of an actor is like designing a communicating finite state machine. Every finite state machine has conceptual states and a set of transitions that change the current internal state. In each conceptual state an actor may accept certain messages, which in turn cause transitions. COOL uses labels to group messages according to their conceptual states. The becomes statement allows an actor to change its conceptual state.

In order to support efficient local error recovery COOL provides a simple form of atomicity. The processing of a message either completes successfully or has no effect. In the later case, all changes made during processing of the current message are undone. After processing a message, an actor must take one of the following primitive actions: commit or becomes when processing is successful and all changes are to be committed; terminate to commit all changes then terminate; or abort when processing is abandoned and all changes are to be discarded. By default, all methods commit unless overridden by an explicit commitment statement.

Timers are an integral part of COOL. An actor may create as many timers as it requires. COOL supports two types of timers: periodic timers and one-shot timers. Each timer has a unique identifier. When a timer expires, it sends a timeout message to its owner.

COOL is not a general purpose programming language and as such, does not provide a rich set of data structuring mechanisms. Every operation in ACube has a corresponding statement or operator in COOL. In COOL run-time exceptions are handled transparently by the run-time system. Exceptions can arise when input messages are accepted, when messages are transmitted, when resources become exhausted, and when the current conceptual state of an actor becomes undefined.
Chapter 6

Validating COOL Specifications

In Chapters 2 and 3 we presented our Timed Actor model to describe actor behaviour which can be modelled by Equation 2.8 (see page 30) and summarized in transition tables similar to those presented in Tables 2.2 and 2.3. Our Timed Actor model allows us to readily specify the behaviour of an actor using a timed actor language like COOL.

In our Timed Actor model several components are required to analyze the behaviour of a system. These include:

1. a specification \((M)\) to model actor behaviour,

2. a notation for specifying the expected behaviour \((\Phi)\) of actors in a system, and

3. a mechanism for comparing the expected behaviour of actors with their observed behaviour in an execution of an implementation \((E)\).

It is possible to analyze such systems using the verification framework provided by a model checker. For a specification \(M\) and a set of system properties \(\Phi_m = \{\phi_1, \phi_2, \ldots, \phi_n\}\) we write

\[
M \models \Phi_m \quad (6.1)
\]

to indicate that the specification \(M\) satisfies \(\Phi_m\). Such analyses are important for verifying the absence of reachable deadlock states in \(M\), for verifying that progress is achieved by specified
processes, and for verifying that specified state properties satisfy the temporal logic formulae expressed by $\Phi_m$. Model checking often deals with qualitative properties but the results of model checking are platform independent.

Our Timed Actor model employs a validation approach for analysis; we compare the observed behaviour of an implementation $E$ created by the COOL compiler with the expected behaviour of an implementation as defined by the checkable properties of the specification. For an implementation $E$ and a set of checkable properties $\Phi_p = \{\phi_1, \phi_2, \cdots, \phi_n\}$ we write

$$E \models \Phi_p$$

(6.2)

to indicate that the implementation $E$ satisfies $\Phi_p$. In this approach $\Phi_p$ is expressed in terms of communication events, the reception and transmission of messages; and describes the expected behaviour and performance of actors in an executable implementation. This class of properties allows a practitioner to ask the following types of questions:

- Does actor $a$ violate invariant $\phi$?
- Can actor $a$ sustain a throughput of $n$ frames per second for message $m$?
- What is the response time of configuration $C$ when actor $a$ issues request $m(x)$ to actor $b$?

Checkable properties are quantitative properties and the results of our validation approach are platform dependent.

In COOL, the performance and coordination properties of actors may be specified and subsequently compiled into an online monitor. This monitor is present in every COOL implementation. At run-time the implementation generates a log of selected actor activity amenable to offline trace analysis. Trace analyzers identify when and where checkable properties are violated.

Monitoring can potentially detect behavioural (protocol) errors, detect timing errors, and validate an implemented system against its specification. Monitoring can also assist in the measurement of selected characteristics of a running system. These include the enumeration of events, estimating the interarrival and intertransmittal times between events, estimating the round trip times of messages, and estimating the response time between events.
Figure 6.1: Steps in validating a COOL specification.

Outline. Figure 6.1 illustrates the key steps required to perform a validation of an implementation. These steps include:

- **Specification**—COOL specifications ($M$) define the behaviour of actors (see Chapter 5). Checkable properties ($\Phi$) describe which actor behaviour to monitor. In Section 6.1 we describe the syntax of COOL checkable properties. In Section 6.2 we describe the semantics of checkable properties, and demonstrate how to specify checkable properties in COOL specifications.

- **Instrumentation**—The COOL compiler translates the definition of actor behaviour into VM instructions ($M'$), and the definition of checkable properties into a set of canonical checkable properties ($\Phi'$). An executable implementation ($E$) is created by binding the VM instructions ($M'$) and the canonical checkable properties ($\Phi'$) to a VM. In Sections 6.3.2–6.3.3 we describe how the COOL compiler uses checkable properties to instrument an implementation, and how the `dprob` clause (see page 111) is used to generate test cases.

- **Monitoring**—An **online monitor** consists of the **monitorable** parts of $M'$ together with the canonical checkable properties $\Phi'$. The VM manages online monitoring. During execution
all monitored activity is stored as an in-memory database at each node running the application. When an application terminates the VM prepares a log of actor activity. Execution may be controlled by application specific run-time parameters. In Section 6.3.1 we describe the structure of online event logs.

- **Analysis**—The *offline trace analyzer* compares the *observed behaviour* of an actor with its *expected behaviour*, as defined by its checkable properties. Analysis requires an application specific *constraint script* to evaluate checkable properties. In Section 6.4 we describe our approach to offline trace analysis.

- **Limitations**—In Section 6.5 we discuss some of the limitations of our approach.

### 6.1 A Syntax and Informal Semantics of Checkable Properties

An actor class specification may contain a special method block labelled `Satisfies \( \notin Q \)` which contains the definition of *checkable properties* to be monitored at run-time. An abstract syntax for checkable property definitions, \( \phi \in Property \), is presented in Figure 6.2. Figure 6.2(a) enumerates the syntactic categories and symbols used in the definition of checkable properties, and Figure 6.2(b) defines the structure of these syntactic categories. The grammar for checkable properties extends the grammar of COOL specifications presented in Section 5.1.

**Events and checkable properties.** *Communication events* form the basis of our approach to validation. For an actor \( a \in C \); a *communications event* \( z \) is either the arrival of message \( m \in \Sigma_C \), denoted \( ?m \), or the transmittal of message \( m \in \Sigma_{actor} \), denoted \( !m \). In COOL specifications, checkable properties defined in class \( C \) can only reference events which occur in class \( C \).
\[ \phi \in \text{Property} \quad \text{the definition of a checkable property} \]
\[ \phi, \phi_1, \phi_2 \quad \text{the name of a checkable property} \]
\[ z, z_1, z_2 \in \text{Events} \quad \text{communication events; message arrival and transmittal} \]
\[ \#z, \#z_1, \#z_2 \in \text{EnumEvents} \quad \text{enumeration events} \]
\[ 'z, "z \in \text{TimedEvents} \quad \text{timed event ‘}z\text{‘ occurs in the current activation} \]
\[ \text{timed event ‘}"z\text{‘ occurs in a past activation} \]
\[ \gamma \in \Gamma \quad \text{an enumeration} \]
\[ \rho \in \Phi(\Gamma) \quad \text{an enumeration constraint} \]
\[ \delta \in \Delta \quad \text{a timed interval} \]
\[ \lambda \in \Phi(\Delta) \quad \text{a timing constraint} \]
\[ v, v', v'' \quad \text{message parameters} \]
\[ s, s', s'' \quad \text{state variables} \]
\[ e \in \text{Exp} \quad \text{an expression (see Figure 5.2)} \]
\[ op \in \text{Op} \quad \text{an operation (see Figure 5.2)} \]
\[ m, m_1, m_2 \in \Sigma\text{actor} \quad \text{the name of a message handler} \]

(a) Syntactic Categories

\[
\begin{align*}
\text{properties} & ::= \text{property}^* \\
\text{property} & ::= \phi \triangleq O \mid \phi \triangleq C \\
O & ::= \rho \mid \lambda \mid z_1 \xrightarrow{n} z_2 \mid m(v) \text{ assert}(expr(v,s)) \\
C & ::= \phi \mid \neg C \mid C_1 \land C_2 \mid C_1 \lor C_2 \\
\gamma & ::= \#z \mid \gamma \text{ op } \gamma_2 \\
\rho & ::= \gamma \mid \gamma \leq e \mid e \leq \gamma \mid \neg \rho \mid \rho_1 \land \rho_2 \mid \rho_1 \lor \rho_2 \\
\delta & ::= 'z_1 \mid "z_2 \\
\lambda & ::= \delta \mid \delta \leq e \mid e \leq \delta \mid \neg \lambda \mid \lambda_1 \land \lambda_2 \mid \lambda_1 \lor \lambda_2 \\
z & ::= ?m \mid !m
\end{align*}
\]

(b) Definitions

Figure 6.2: An abstract syntax of checkable properties, \( \phi \in \text{Property} \).

The following rules (see Figure 6.2) introduce the definition of a checkable property:

\[
\begin{align*}
\text{properties} & ::= \text{property}^* \quad (6.3) \\
\text{property} & ::= \phi \triangleq O \mid \phi \triangleq C \quad (6.4) \\
O & ::= \rho \mid \lambda \mid z_1 \xrightarrow{n} z_2 \mid m(v) \text{ assert}(expr(v,s)) \quad (6.5) \\
C & ::= \phi \mid \neg C \mid C_1 \land C_2 \mid C_1 \lor C_2 \quad (6.6)
\end{align*}
\]

The definition of a simple checkable property takes the form \( \phi \triangleq O \) (Rule 6.4), where \( \phi \) is the name of a checkable property definition, and \( O \) is one of the following types of checkable properties.
(Rule 6.5):

- An enumeration constraint $\rho$ is used to collect information on the frequency of communication events (see Section 6.2.1). Enumeration constraints are defined in terms of enumerations of the form $\#z$.

$$\phi_1 \triangleq 0 \leq (\#?put - \#!overflow) - \#!out \leq n$$

- A timing constraint $\lambda$ is used to measure the interval of time between two communications events $z_1$ and $z_2$. Timing constraints are defined in terms of timed intervals of the form $'z_1 - "z_2$ (see Section 6.2.2).

$$\phi_2 \triangleq (\mu - \sigma) \leq '!out - "!out \leq (\mu + \sigma)$$
$$\phi_3 \triangleq 0 \leq '?put - "?put \leq (\mu \times (n - 1))$$

- A follows constraint $z_1 \rightsquigarrow z_2 : n$ is used to determine if two causally related communication events, $z_1$ and $z_2$, occur within a specified period of time $n$ (see Section 6.2.3).

$$\phi \triangleq !ho\_request \rightsquigarrow ?ho\_request\_ack : 400 \text{ usec}$$

The term $z_1 \rightsquigarrow z_2$ is referred to as a follows event.

- A message precondition ‘m(v) assert(expr(v, s))’ evaluates an assertion $expr(v, s)$ each time message $m$ is activated. The assertion may contain references to message parameters ($v$) and state variables ($s$) (see Section 6.2.4).

$$\phi \triangleq \text{data(seq,x) assert(current == Act)}$$

Rule 6.6 indicates that complex checkable properties can be defined using simple checkable properties and logical connectives; for definitions $\phi_1$, $\phi_2$, and $\phi_3$ given above we may write,

$$\phi_4 \triangleq (\phi_1 \land \phi_2) \lor \phi_3$$
Correspondence between $\Phi_p$ and $\Phi_m$. It is interesting to note (see Table 6.1) that many checkable properties ($\Phi_p$) are expressible as temporal logic formulae ($\Phi_m$); that is $\Phi_m \cap \Phi_p \neq \emptyset$.

Table 6.1: Checkable properties that can be expressed as temporal logic formulae.

<table>
<thead>
<tr>
<th>Checkable Property ($\Phi_p$)</th>
<th>Temporal Logic Formula ($\Phi_m$)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m(v) ; \text{assert}_{\text{expr}(v,s)}$</td>
<td>$\square p, ; p = \text{expr}(s,v)$</td>
<td>Property $p$ remains invariantly true.</td>
</tr>
<tr>
<td>$z_1 \Rightarrow z_2$</td>
<td>$\square (z_1 \Rightarrow \Diamond z_2)$</td>
<td>Property $z_1$ always implies eventually $z_2$.</td>
</tr>
<tr>
<td>$n \leq #z_1 - #z_2 \leq m$</td>
<td>$\square p, ; p = (n \leq #z_1 - #z_2 \leq m)$</td>
<td>Property $p$ remains invariantly true.</td>
</tr>
<tr>
<td>$n \leq 'z_1 - ''z_2 \leq m$</td>
<td>Not representable in temporal logic.</td>
<td></td>
</tr>
</tbody>
</table>

6.2 Specifying Checkable Properties

Enumeration events, timed events, and follows events are defined in terms of communication events:

\[ z \; ::= \; ?m \; | \; !m \]  

where $?m$ denotes a reception event, and $!m$ denotes a transmission event.

6.2.1 Enumeration Events and Constraints

An enumeration event $\#z$ requests the offline trace analyzer (see Section 6.4) to determine either the number of times message $m$ is received ($\#?m$), or the number of times message $m$ is transmitted ($\#!m$).

**Definition 6.1 (Enumeration Constraints)** Let $\Gamma$, with typical elements $\gamma, \gamma_1, \gamma_2, \cdots$, be a finite set of enumerations each defined inductively by

\[ \gamma \; ::= \; \#z \; | \; \gamma_1 \; \text{op} \; \gamma_2 \]  

where $\text{op} \in \{+,-,\text{div}, \times\}$. Then the set $\Phi(\Gamma)$ of enumeration constraints $\rho$ is defined inductively by

\[ \rho \; ::= \; \gamma \; | \; \gamma \leq e \; | \; e \leq \gamma \; | \; \neg \rho \; | \; \rho_1 \land \rho_2 \; | \; \rho_1 \lor \rho_2 \]
where constant $e \in \mathbb{N}$.

**Figure 6.3:** A message sequence chart illustrating competition for a distributed binary lock with timeout.

**Example—Monitoring the binary lock invariant of a distributed lock implementation.** The interactions between a distributed binary lock $d \in \text{DLock}$ and two actors competing for the lock is illustrated in Figure 6.3. Counting events $\text{!locked}$ and $\text{!unlocked}$ allows us to determine if the binary
lock invariant is ever violated;

\[ \phi \triangleq 0 \leq \#!\text{locked} - \#!\text{unlocked} \leq 1 \]  

(6.10)

at any instant, the number of available locks must be 0 or 1.

Assume there is an active distributed lock \( d \in \text{DLock} \). Any actor may request the lock by sending the message \( \text{lock}() \Rightarrow d \) which has two outcomes. If \( d \) responds with message \( \text{locked}() \); the lock is acquired \((b_1)\), or the requesting actor already holds the lock \((b_2)\). In all other cases \( d \) responds with message \( \text{reject}() \) indicating that the caller did not acquire the lock \((b_3)\).

Once acquired the lock can only be held for a fixed period of time. This prevents starvation amongst actors competing for the lock. This policy is enforced by timer \( t \). When timer \( t \) expires lock \( d \) sends message \( \text{unlocked}() \) to the actor holding the lock \((b_4)\).

The actor holding the lock may release the lock by sending the message \( \text{unlock}() \Rightarrow d \) which has two outcomes. If \( d \) responds with message \( \text{unlocked}() \), the lock is released \((b_5)\). In all other cases \( d \) responds with message \( \text{reject}() \) indicating that the caller is not the current holder the lock.

6.2.2 Timed Events and Local Timing Constraints

Figure 6.4 illustrates a sequence of communication events experienced by an actor \( a \). Events local to an actor are totally ordered and every event can unambiguously be assigned to an activation. Activations always begin with a reception event \( ?m \), and there can only be a single reception \( ?m \) in each activation. In Figure 6.4 activations occur at values 100, 110, and 118 on the local logical clock \((L)\) accessible by actor \( a \). We refer to the activation at clock value 118 as the current activation, as actor \( a \) is currently processing message \( m_1 \); any event \( z \) occurring in the current activation is denoted \( 'z \).

In a similar fashion, the activations beginning at clock values 100 and 110 are referred to as past activations. Any event \( z \) occurring in a past activation is denoted \( "z \). This creates some ambiguity, for example in Figure 6.4, the transmission events for message \( m_2 \) at clock values 104 and 112 can both be denoted \( "!m_2 \). For the purpose of analysis (see Section 6.4), when we compute the timed interval between event \( z_1 \) in the current activation and event \( z_2 \) in a past activation, denoted
\[ \delta = 'z_1 - ''z_2, \text{ we are only concerned with most recent past occurrence of event } z_2. \]

In summary, reception events \( ?m \) divide an actor's execution into a finite number of activations. The event \( '?m \) refers to the reception event which started the current activation. An event \( ''z \) refers to the most recent past communication event \( z \). This leads us to the following proposition:

**Proposition 6.1 (Timed Intervals)**

All timed intervals reference the local master-clock \( (\mathbb{T}) \). The timed interval between any two communication events \( z_1 \) and \( z_2 \) in the current activation is zero. The timed interval between any two communication events \( z_1 \) and \( z_2 \) in distinct activations is greater than zero.

\[ 'z_1 - 'z_2 = 0, \forall z_1, z_2 \quad (6.11) \]
\[ 'z_1 - ''z_2 \geq 0, \forall z_1, z_2 \quad (6.12) \]

In COOL, checkable properties referencing timed intervals are most commonly expressed as timing constraints. Our notion of timing constraints is influenced by the definition of clock constraints in Alur and Dill [see 52, Definition 3.6].

**Definition 6.2 (Timing Constraints)** Let \( \Delta \), with typical elements \( \delta, \delta_1, \delta_2, \cdots \), be a finite set of timed intervals each of the form \( 'z_1 - ''z_2 \). Then the set \( \Phi(\Delta) \) of timing constraints \( \lambda \) is defined inductively by

\[ \lambda ::= \delta \mid \delta \leq e \mid e \leq \delta \mid \neg \lambda \mid \lambda_1 \land \lambda_2 \mid \lambda_1 \lor \lambda_2 \quad (6.13) \]

where constant \( e \in \mathbb{Q} \).
In COOL we employ timed intervals of the form $\delta = z_1 - z_2'$ in all timing constraints, where events $z_1$ and $z_2$ occur in distinct activations of an actor $a$. Figure 6.5 illustrates the four possible timed intervals of a simple timing constraint involving events $z_1'$ and $z_2''$.

Figure 6.5: An interpretation of timed intervals; all timed intervals are local to actor $a$. 
An interval of the form $\delta = \langle ?m_1 - "?m_2 \rangle$ (see Figure 6.5(a)) is used to determine **interarrival times** between messages $m_2$ and $m_1$. When $m_1 = m_2$, monitoring interarrival times allows us to estimate interarrival rates of a message $m$.

An interval of the form $\delta = \langle !m_1 - "?m_2 \rangle$ (see Figure 6.5(b)) is used to determine **turnaround times**—the time between accepting a request $m_2$ and generating a response $m_1$. If events $!m_1$ and $?m_2$ occur in the same activation ($\langle !m_1 - "?m_2 \rangle$), $\delta = 0$.

An interval of the form $\delta = \langle ?m_1 - "!m_2 \rangle$ (see Figure 6.5(c)) is used to determine **response times**—the time between submitting a request $m_2$ and receiving a response $m_1$. If events $?m_1$ and $!m_2$ occur in the same activation ($\langle ?m_1 - "!m_2 \rangle$, $\delta$ would be undefined.

An interval of the form $\delta = \langle !m_1 - "!m_2 \rangle$ (see Figure 6.5(d)) is used to determine **intertransmittal times** between messages $m_2$ and $m_1$. When $m_1 = m_2$, monitoring intertransmittal times allows us to estimate intertransmittal rates of a message $m$. If events $!m_1$ and $!m_2$ occur in the same activation ($\langle !m_1 - "!m_2 \rangle$, $\delta = 0$.

**Example—Monitoring the performance of an n-ary buffer.** In the distributed buffer class Buffer (Listing D.2 on page 229) the timing constraint

$$\phi_2 \triangleq (\mu - \sigma) \leq \langle !out - "out \rangle \leq (\mu + \sigma)$$

measures the intertransmittal times of event $!out$ from the perspective of actor $b \in Buffer$. We expect intertransmittal times to fall in the range $(\mu - \sigma) \leq \delta_1 \leq (\mu + \sigma)$ time units, where $\mu$ is the mean transmittal time of message $out$ and $\sigma$ is the acceptable jitter in the mean transmittal times of message $out$.

In a similar fashion the timing constraint

$$\phi_3 \triangleq 0 \leq \langle ?put - "?put \rangle \leq (\mu \times (n - 1))$$

measures the interarrival times of event $?put$ from the perspective of actor $b \in Buffer$. We expect interarrival times of event $?put$ to fall in the range $0 < \delta_2 \leq (\mu \times (n - 1))$ time units, where $\mu$ is the mean arrival time of message $put$ and $n$ is the capacity of the buffer.
The interaction between a producer \((p \in \text{Producer})\), a consumer \((c \in \text{Consumer})\), and a buffer \((b \in \text{Buffer})\) is illustrated in Figure 6.6. When the buffer is in state \text{Empty} a \text{get()} request from a consumer is handled with an \text{underflow()} response from actor \(b\). In a similar fashion, when the buffer
buffer is in state **Full** a put(x) request from a producer is handled with a overflow() response from actor b. When the buffer is in state **PartFull** or **Full** a get() request from the consumer is handled by an out(x) response from actor b. When constraints $\phi_2$ and $\phi_3$, and the $n$-ary buffer invariant ($\phi_1$) are satisfied, actor b is able to deliver a stream of buffers to the consumer with an arrival time of $\mu \pm \sigma$ time units.

### 6.2.3 Follows Constraints

A *follows constraint* of the form

$$\phi \triangleq z_1 \rightsquigarrow z_2 : n$$

asserts that event $z_2$ *follows* event $z_1$ within $n$ time units. Follows constraints allow us to monitor timing properties between causally related events within the same actor instance. A follows constraint of the form $z_1 \rightsquigarrow z_2 : n$ is equivalent to the timed interval $0 \leq t_{z_2} - t_{z_1} \leq n$.

**Example—Monitoring causally related events.** In this section we complete the description of the vending machine configuration described in Sections 2.4 and 3.10, and illustrated in Figure 2.4. A COOL specification for this vending machine configuration is presented in Listing D.1 on page 228. In the specification a vending machine user must make an item selection and enter the correct coinage within 15 seconds.

$$\phi_1 \triangleq \textnormal{?init} \rightsquigarrow \textnormal{?dispense} : 15 \textnormal{ sec}$$

(6.17)

In this application timer $t$ is used to enforce a 15 second timeout, while constraint $\phi_1$ collects information for each use of the vending machine. Failure to enter the correct coinage within 15 seconds results in the restart of the vending machine. The information collected with $\phi_1$ is important in determining the period of timer $t$.

### 6.2.4 Message Preconditions

A *message precondition* of the form

$$\phi \triangleq m(v,v',\cdots) \textnormal{ assert}(expr(v,v',\cdots,s,s',\cdots))$$

(6.18)
evaluates the assertion \( expr(v, v', \ldots, s, s', \ldots) \) involving message parameters \( v, v', \ldots \) and state variables \( s, s', \ldots \) each time message \( m \) is received. Message preconditions are the only checkable properties requiring evaluation at run-time.

Figure 6.7: A message sequence chart illustrating the interaction between a sender \( s \in \text{Sender} \) and a receiver \( r \in \text{Receiver} \) in alternating bit protocol.

Example—Monitoring lost and corrupt messages in alternating bit protocol. The definition of class \( \text{Receiver} \) in the specification of alternating bit protocol (Listing D.4 on page 231) employs a message precondition of the form

\[
\phi_3 \triangleq f(seq, x) \text{ assert}(fe == seq)
\]
to identify situations when either message \( f() \) is corrupt, or an acknowledgement \( g() \) is lost. In this implementation state variable \( fe \) indicates the sequence number of the next data frame the receiver is expecting (either 0 or 1), and message parameter \( seq \) indicates the sequence number of the current data frame (either 0 or 1).

The interaction between a sender \((s \in \text{Sender})\) and a receiver \((r \in \text{Receiver})\) is illustrated in Figure 6.7. Initially, \( s \) transmits data frame \( f(0,x) \) which is received by \( r \) and acknowledged with control frame \( g(0) \). At this point \( \phi_3 \) evaluates to \text{true} (labelled as \( fe = seq \) in Figure 6.7). Actor \( s \) interprets the arrival of control frame \( g(0) \) as an acknowledgement enabling it to transmit the second data frame \( f(1,x) \). This is received by \( r \) and acknowledged with control frame \( g(1) \), however, this acknowledgement is lost. Timer \( t \) is used by \( s \) to recover from lost acknowledgements. When timer \( t \) expires it sends message \( \text{noack}(seq,x) \) to \( s \) which forces a retransmission of its last data frame, in this case it retransmits \( f(1,x) \). Actor \( r \) interprets this data frame as a duplicate and retransmits its last control frame \( g(1) \). At this point \( \phi_3 \) evaluates to \text{false}. Actor \( s \) interprets \( g(1) \) as an acknowledgement and prepares to send the next frame.

### 6.3 Instrumenting a COOL Implementation

In this section we provide a brief description of our monitoring approach (see Section 6.3.1), and a brief description of how the COOL compiler \textit{instruments} a COOL implementation. A more complete treatment of the translational semantics of COOL can be found in [see 14, Section 4.2.4 on pages 78–82 for typechecking; and Section 5.2.3 on pages 101–112, and Section 5.2.4 on pages 113–119 for code generation].

\textbf{Translation strategy.} We illustrate the translation of selected COOL language features to VM instructions using a translation of the form

\[ [S] \mapsto \text{emit}[L] \quad (6.20) \]

where \( S \) is a feature in the source language, in this case COOL; \( L \) are the equivalent VM instructions for implementation language \( L \); and \( \mapsto \) describes the translation semantics from \( S \) to \( L \) [see 14,
Chapters 4 and 5]. All text between the delimiters [ and ] is text in the source language, and all text between the delimiters emit[ and ] is text in the implementation language.

6.3.1 Monitoring a COOL Implementation

An actor class $C$ is *monitorable* when its specification includes definitions for checkable properties (see Section 6.2). Every actor instance $a \in C$ of a monitored class $C$ is subject to run-time monitoring. Monitoring the activity of an actor involves *logging* selected events to an *event log*.

The structure of a COOL event log is presented in Figure 6.8. Figure 6.8(a) describes the elements of an event log, and Figure 6.8(b) defines the general structure of an event log. Event logs with this structure are amenable to the type of offline trace analysis described in Section 6.4. An event log records the sequence (*trace*) of activations performed by an application. There are four types of activations: $(b)$ initializes an application, $(k)$ activates an actor and records selected events, $(j)$ activates a timer and records the details of a timeout (the run-time system logs the details of timeouts $(j)$), and $(f)$ finalizes an application and stores the event log.

Let $Y$, with typical elements $y, y', \ldots$, be a finite set of event types generated by an active actor. An actor activation $(k)$ is a sequence of events of type $y \in Y$: $l$ logs the details of a dropped message, $r$ logs the details of a receive operation, $p$ logs the result of evaluating a message precondition, $n$ logs the details of a *new* operation, $t$ logs the details of a *trigger* operation, $d$ logs the details of a *discard* operation, $s$ logs the details of a transmission operation, $c$ logs the details of a commitment operation, and $e$ logs the details of a run-time exception.

The COOL compiler instruments those operations (*monitorable code*) in a specification that are referenced by the checkable properties; events of type ‘$p$’ reference message preconditions, events of type ‘$r$’ reference message receptions, and events of type ‘$s$’ reference message transmissions. In addition, the COOL compiler instruments $dprob$ clauses (‘$l$’), commitment statements (‘$c$’), and run-time exceptions (‘$e$’). These events are often required to understand the results of an analysis (see Section 6.4). Collectively, all monitorable code in an instrumented application together with the canonical checkable properties ($\Phi'$), is referred to as an *online monitor*. 
(b) initialize application (see Section C.1.1)
(k) actor activation (dispatch)
(j) timer activation (timeout) (see Section C.1.2)
(f) finalize application

Event types: \( y \in Y = \{l, r, p, n, t, d, s, c, e\} \)

- \( l \) drop message event (see Section C.1.3)
- \( r \) a receive event (see Section C.1.4)
- \( p \) evaluate a message precondition (see Section C.1.5)
- \( n \) a new operation (see Section C.1.6)
- \( t \) a trigger operation (see Section C.1.7)
- \( d \) a discard operation (see Section C.1.8)
- \( s \) a transmission operation (see Section C.1.9)
- \( c \) a commitment operation (see Section C.1.10)
- \( e \) a run-time exception (see Section C.1.11)

(a) Elements of a COOL Event Log

\[
\text{Log} ::\ = \ \left[ \langle b \rangle \ (\langle k \rangle \ | \ (j) \right)^* \ (f) \right] \\
\langle k \rangle ::\ = \ \langle l \rangle \ | \ \langle r \ p^* \ (n \ | \ t \ | \ d \ | \ s \ | \ e \ | \ c \rangle \right)
\]

(b) COOL Event Log Structure

Figure 6.8: The structure of a COOL event log.

Instrumentation is performed in two stages. In the first stage checkable properties are analyzed by the type checker to identify monitorable operations (see Section 6.3.2). In the second stage the code generator produces monitoring code (see Section 6.3.3). Every segment of monitorable code generated by the COOL compiler references a logging function \( \text{Log}(y, \cdots) \). At run-time, the execution of statement \( \text{Log}(y, \cdots) \) creates a unique event log entry of type \( y \). For example, if an implementation of class \( C \) contains a monitorable message handler \( m \), a log record of type ‘r’ will be generated each time message \( m \) is activated, denoted \( \text{Log}(‘r’, \cdots) \). Each event type has a unique context, denoted here by \( \cdots \), which is described in Section C.1.

\(^1\)Monitoring of these operations is not required for the evaluation of checkable properties.
### 6.3.2 Identifying Monitorable Operations

Algorithm **FindMonitorable** (Listing 6.2) illustrates how the COOL compiler analyzes checkable properties to identify and classify monitorable operations. Algorithm **FindMonitorable** identifies assertions and communication events of the form !\(m\) and ?\(m\). Events of the form #\(z\), ', \(z\) ', "\(z\), and \(z_1 \Rightarrow z_2\) are analyzed by the **offline trace analyzer** (see Section 6.4).

The algorithm requires three data sets for each monitorable class (see Table 6.2); \(\Phi_C\), the set of simple checkable properties for class \(C\); \(\Sigma_C\), the set of input messages for class \(C\); and \(\Pi_C\), the dictionary of output messages for class \(C\). For each distinct transmission operation !\(m\) found in a specification, a dictionary reference \(\Pi_C(m)\) returns the set of message handlers which issue !\(m\), denoted \(m \mapsto \{m_1, m_2, \ldots, m_k\}\).

**Table 6.2:** Data sets required to instrument actor class \(C\).

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Phi_C = {\phi_1, \phi_2, \ldots, \phi_n})</td>
<td>Checkable properties.</td>
</tr>
<tr>
<td>(\Sigma_C = {m_1, m_2, \ldots, m_n})</td>
<td>Input messages.</td>
</tr>
<tr>
<td>(\Pi_C = {m_o_1 \mapsto {m_1, m_2, \ldots, m_k}, \ldots, m_o_n \mapsto {m_1, m_2, \ldots, m_k}})</td>
<td>Dictionary of output messages.</td>
</tr>
<tr>
<td>(A_C = \begin{bmatrix} m_1 \mapsto {(\phi_1, expr_1(v,s)), \ldots, (\phi_k, expr_k(v,s))}, \ldots, m_n \mapsto {(\phi_1, expr_1(v,s)), \ldots, (\phi_k, expr_k(v,s))}\end{bmatrix})</td>
<td>Dictionary of assertions.</td>
</tr>
<tr>
<td>(\Upsilon_C = {m_1, m_2, \ldots, m_n})</td>
<td>Monitorable message handlers.</td>
</tr>
<tr>
<td>(\Psi_C = {m_o_1 \mapsto {m_1, m_2, \ldots, m_k}, \ldots, m_o_n \mapsto {m_1, m_2, \ldots, m_k}})</td>
<td>Dictionary of monitorable output messages.</td>
</tr>
</tbody>
</table>

Algorithm **FindMonitorable** produces three data sets for each monitorable class; \(A_C\), the dictionary of assertions defined for class \(C\); \(\Upsilon_C \subseteq \Sigma_C\), the set of monitorable message handlers in class \(C\); and \(\Psi_C \subseteq \Pi_C\), the dictionary of monitorable output messages in class \(C\). For each message handler \(m\), a dictionary reference \(A_C(m)\) returns the list of assertions defined for message handler \(m\), denoted \(m \mapsto \{(\phi_1, expr_1(v,s)), \ldots, (\phi_k, expr_k(v,s))\}\). For each monitorable output message \(m\), a dictionary reference \(\Psi_C(m)\) returns the set of message handlers which issue !\(m\), denoted
\[ m \mapsto \{ m_1, m_2, \cdots, m_k \}. \]

**Details of algorithm FindMonitorable.** Algorithm FindMonitorable examines each simple checkable property \( \phi \in \Phi_C \) to identify assertions and communication events. If \( \phi \) is an assertion of the form ‘\( m(v) \text{ assert}(\text{expr}(v,s)) \)’ message \( m \) is added to the list of monitorable message handlers \( \Upsilon_C \), and the assertion is added to the dictionary of assertions \( A_C \mapsto [m \mapsto (\phi, \text{expr}(v,s))] \). When \( \phi \) contains a reception event of the form \( ?m \), message \( m \) is added to the list of monitorable message handlers \( \Upsilon_C \). When \( \phi \) contains a transmission event of the form \( !m \), the compiler prepares a list of message handlers which transmit \( m \) by examining \( h \leftarrow \Pi_C(m) \), and adds this list of message handlers to \( \Upsilon_C \). The list of message handlers must also be added to the dictionary of output messages \( \Psi_C \mapsto [m \mapsto \{ h \}] \).

### 6.3.3 Instrumenting a Message Handler

Instrumentation produces an implementation capable of monitoring selected assertions and communication events at run-time. Algorithm FindMonitorable (see Listing 6.2), run as part of the COOL type checker, provides the COOL code generator with the data sets \( (A_C, \Upsilon_C, \Psi_C) \) necessary to identify the monitorable operations for each actor class.

Algorithm Instrument (Listing 6.2) illustrates how a message handler \( m \) is instrumented for online monitoring. We describe the generation of monitorable code using the translation scheme described by Equation 6.20 on page 142. If statement \( s \) is not monitorable, it is translated to VM instructions using function Generate\( (s) \). Algorithm Instrument instruments the following COOL language features:

- **InstrumentDprob\( (m, p) \)**—If the definition of message handler \( m \) contains a \texttt{dprob} clause, the code generator emits the following monitoring code, as a prologue to \( m \), to handle drop probabilities.
Function `chance(p)` returns `true` with probability $p$, and `false` with probability $1 - p$, for $0.0 \leq p \leq 1.0$. When message handler $m$ is activated, function `chance(p)` draws a sample $y$ from the uniform distribution $U(0, 1)$. When $y > p$ `chance(p)` returns `false`; otherwise when $y \leq p$ `chance(p)` returns `true`, and the monitored actor aborts the current activation.

The `dprob` clause automatically generates test cases.

In monitorable code function $L()$ returns the current value of $L$, the system logical clock (see Chapter 4 on page 75).

- **InstrumentReceive($m$)**—If $m$ is a monitorable message handler, the code generator emits the following monitoring code to record an actor activation. In this record, $\star$ is the identity of the actor sourcing message $m$, $\

- **InstrumentAssertion($m, \phi, expr(v,s)$)**—If $m$ is a monitorable message handler and the definition of $m$ contains a message precondition $\phi$, the code generator emits the following monitoring code to evaluate assertion $expr(v,s)$ at run-time. Function `_fassert(expr(v,s))` returns the result of evaluating assertion $expr(v,s)$. The result of an evaluation is logged only when an assertion fails.
• InstrumentTransmit\((a, f, v)\)—If \(m\) is a monitorable message handler, and \(f\) is a monitorable output message, the code generator emits the following monitoring code to record the transmission.

\[
\begin{align*}
[f(v) \Rightarrow a] & \quad \rightarrow \quad \text{emit} \\
& \quad \quad \text{Log('s', L()+1, f, a);}
& \quad \quad b \leftarrow \text{vmsend(self, a, f);} \\
& \quad \quad b.v \leftarrow v;
\end{align*}
\]

• InstrumentCommit\((r)\)—If \(m\) is a monitorable message handler, the code generator emits monitoring code to match the commitment operator \(r\) which ended message processing.

\[
\begin{align*}
\{ \text{commit} \} & \quad \rightarrow \quad \text{emit} \\
& \quad \quad \text{Log('c', L()+1, commit, current);} \\
& \quad \quad \text{vmbecomes(current);} \\
\{ \text{becomes } q \} & \quad \rightarrow \quad \text{emit} \\
& \quad \quad \text{Log('c', L()+1, becomes, q);} \\
& \quad \quad \text{vmbecomes(q);} \\
\{ \text{terminate} \} & \quad \rightarrow \quad \text{emit} \\
& \quad \quad \text{Log('c', L()+1, terminate, current);} \\
& \quad \quad \text{vmterminate();} \\
\{ \text{abort} \} & \quad \rightarrow \quad \text{emit} \\
& \quad \quad \text{Log('c', L()+1, abort, current);} \\
& \quad \quad \text{vmabort();}
\end{align*}
\]

6.3.4 Example—Instrumenting an \(n\)-ary Buffer

The distributed buffer class Buffer (see Listing D.2 on page 229) defines the following set of checkable checkable properties \(\Phi_{\text{Buffer}}\).

\[
\begin{align*}
\phi_1 & \triangleq 0 \leq (?_\text{put} - !_\text{overflow}) - !_\text{out} \leq n \\
\phi_2 & \triangleq (\mu - \sigma) \leq !_\text{out} - !_\text{out} \leq (\mu + \sigma) \\
\phi_3 & \triangleq 0 \leq !_\text{put} - !?_\text{put} \leq (\mu \times (n - 1)) \\
\phi_4 & \triangleq (\phi_1 \land \phi_2) \lor \phi_3
\end{align*}
\]

From \(\Phi_{\text{Buffer}}\) algorithm \text{FindMonitorable} identifies the following monitorable communication events.
\[
\phi_1 \supseteq \{ ? \text{put}, ! \text{overflow}, ! \text{out} \} \\
\phi_2 \supseteq \{ ! \text{out} \} \\
\phi_3 \supseteq \{ ? \text{put} \}
\]

When we apply algorithm \texttt{FindMonitorable} to conceptual state \texttt{Full} of the distributed buffer class \texttt{Buffer} we compute the following data sets.

\[
\Phi_{\text{Buffer}} = \{ \phi_1, \phi_2, \phi_3 \} \\
\Sigma_{\text{Buffer}} = \{ \text{init()}, \text{put()}, \text{get()}, \text{stop()} \} \\
\Pi_{\text{Buffer}} = \{ \text{out()} \mapsto \{ \text{get()} \}, \text{overflow()} \mapsto \{ \text{get()} \}, \text{underflow()} \mapsto \{ \text{put()} \} \} \\
A_{\text{Buffer}} = [ ] \\
\Upsilon_{\text{Buffer}} = \{ \text{get()}, \text{put()} \} \\
\Psi_{\text{Buffer}} = \{ \text{out()} \mapsto \{ \text{get()} \}, \text{overflow()} \mapsto \{ \text{get()} \} \}
\]

Applying algorithm \texttt{Instrument} to data sets \(A_{\text{Buffer}}, \Upsilon_{\text{Buffer}}\) and \(\Psi_{\text{Buffer}}\) we arrive at the instrumentation of message handlers \texttt{get()} and \texttt{put()} illustrated in Listing 6.1. An activation of message handler \texttt{get} produces two possible traces,

\[
T_1 = \langle \\
\quad \begin{array}{l}
\quad r : (L, \text{self}, \text{get}, \ast, \text{now}, \text{Full}) \\
\quad s : (L + 1, \text{out}, s,c) \\
\quad c : (L + 1, \text{becomes}, \text{Empty})
\end{array} \\
\rangle
\]

\[
T_2 = \langle \\
\quad \begin{array}{l}
\quad r : (L, \text{self}, \text{get}, \ast, \text{now}, \text{Full}) \\
\quad s : (L + 1, \text{out}, s,c) \\
\quad c : (L + 1, \text{becomes}, \text{PartFull})
\end{array} \\
\rangle
\]

depending on the state of the buffer after \texttt{get()} request is serviced.

An activation of message handler \texttt{put} produces two possible traces,
\[ T_1 = \{ \]
\[ l: (L, self, put) \]
\[ c: (L + 1, abort, Full) \]
\[ \} \]
\[ T_2 = \{ \]
\[ r: (L, self, put, *, now, Full) \]
\[ s: (L + 1, overflow, s, p) \]
\[ c: (L + 1, becomes, Full) \]
\[ \} \]

depending on whether the input message is dropped by a `dprob` clause.

---

**Listing 6.1: Instrumenting message handlers `get` and `put`**

```
// Uninstrumented COOL specification

Full:

msg get() from c {
  out(deQueue()) => c;
  if (isEmptyQ() == true)
    becomes Empty;
  else
    becomes PartFull;
}

msg put(int x) from p dprob 0.01 {
  overflow(x) => p;
  commit;
}

// Instrumented VM instructions

Full:

case Buffer_get: if (source(m) == s.c) {
  Log('r', L(), self, get, *, now, Full);
  Log('s', L()+1, out, s.c);
  b ← vmsend(self, s.c, out);
  b.x ← deQueue();
  if (isEmptyQ() == true) {
    Log('c', L()+1, becomes, Empty);
    vmbecomes(Empty);
  } else {
    Log('c', L()+1, becomes, PartFull);
    vmbecomes(PartFull);
  }
  break;
}
```

```
case Buffer_put: if (source(m) == s.p) {
  if (chance(0.01) == true) {
    Log('l', L(), self, put);
    Log('c', L()+1, abort, Full);
    vmabort();
  } else {
    Log('c', L()+1, becomes, PartFull);
    vmbecomes(PartFull);
  }
  break;
}
```

```
Log('r', L(), self, put, *, now, Full);
Log('s', L()+1, overflow, s.p);
b ← vmsend(self, s.p, overflow);
```

```
Log('c', L()+1, becomes, Full);
vmbecomes(current);
```
Listing 6.2: Algorithm FindMonitorable—finding monitorable communication events; and Algorithm Instrument—instrumenting a message handler.

```plaintext
algorithm FindMonitorable(Φ_C, Σ_C, Π_C)
  Y_C = {}, Ψ_C = [], A_C = []
  for each φ ∈ Φ_C
    if φ is an assertion of the form 'm(v) assert(expr(v,s))'
      Y_C = Y_C ∪ m
      A_C = [m → (φ, expr(v,s))]
    else
      for each ?m in φ
        do {Y_C = Y_C ∪ m}
      then
        for each !m in φ
          do {handlers = Π_C(m)
               then
                 Y_C = Y_C ∪ handlers
               } do
          if A_C(m) = φ
            then
              // Generate code for an assertion record
              list ← A_C(m)
              for each (φ, expr(v,s)) ∈ list
                do {InstrumentAssertion(m, φ, expr(v,s))}
          return (Y_C, Ψ_C, A_C)

algorithm Instrument(m, p, Y_C, Ψ_C, A_C)
  if m = message handler, p = drop probability for m
  if p ≠ 0.0
    then { // Generate code for a drop record
      InstrumentDrop(m, p)
    if m ∈ Y_C
      then { // Generate code for a receive record
        InstrumentReceive(m)
      if A_C(m) = φ
        then
          // Generate code for an assertion record
          list ← A_C(m)
          for each (φ, expr(v,s)) ∈ list
            do {InstrumentAssertion(m, φ, expr(v,s))}
      return

algorithm Instrument(continued)
  for each monitorable statement s in message handler m
    if f ∈ Ψ_C ∧ m ∈ Ψ_C(f)
      then { // Generate a monitorable transmission statement
        InstrumentTransmit(a, f, v)
      else
        { // Generate VM transmission operator
          emit(b ← vmsend(self, a, f))
          emit[h.v ← v]
        do case s
          if m ∈ Y_C
            then { // Generate a monitorable commitment statement
              InstrumentCommit(r)
            else
              { // Generate a VM commitment operator
                then
                  case r
                    of { commit → emit[vmbecomes[current]]
                    becomes q → emit[vmbecomes(q)]
                    terminate → emit[vmterminate()]
                    abort → emit[vmabort()]
                otherwise:
                  Generate(s)
```

The text contains algorithms for finding monitorable communication events and instrumenting a message handler. The algorithms involve processing assertions, handling transmission statements, and committing statements. The text is formatted in a structured manner, with clear delineation of code sections and comments explaining the logic. The algorithms are designed to be modular and readable, with clear uses of variables and conditions to control the flow of the code.
6.4 Offline Trace Analysis

In our approach, an implementation is validated by comparing the observed behaviour of monitored actors with their expected behaviour, as defined by their checkable properties. In Figure 6.9 we illustrate the steps required to analyze the monitoring results for an implementation $E$. These steps include:

1. Obtaining the COOL event logs—The observed behaviour of the monitored actors executing implementation $E$ on nodes $n_1, n_2, \cdots, n_k$ is provided in the form of COOL event logs (see Section 6.3.1).

2. Preparing the trace database—Application cdb collates the $k$ event logs into a single trace database $T = \{T_1, T_2, \cdots, T_k\}$, where $T_i$ contains the trace records of node $i$. The structure of trace database records is described in [13, Section B.3 on page 111]. Trace databases prepared by cdb are partitioned by node and sorted on field clock which contains the value of the local logical clock when record was collected.

3. Performing a trace analysis—The offline trace analyzer processes a trace database and pro-

![Diagram](image_url)
vides a summary of the observed behaviour of monitored actors, and a variance report indicating when observed behaviour does not agree with the expected behaviour. To perform these tasks the trace analyzer requires two additional pieces of information:

- $\Phi'$—the set of canonical checkable properties which contain the information required by a trace analyzer to parse a trace database, and
- Evaluate()—an application specific constraint script which compares observed and expected behaviour.

**Canonical checkable property.** A canonical checkable property $\phi'$ is a simple checkable property with constant constraints, arithmetic operators, and logical connectives removed. A canonical checkable property retains only the checkable terms required for online monitoring. For example, the enumeration constraint for class Buffer in Listing D.2 on page 229

$$\phi_1 \triangleq 0 \leq (#\text{? put} - #!\text{overflow}) - #!\text{out} \leq n \quad (6.21)$$

has a canonical representation $\phi'_1$.

<table>
<thead>
<tr>
<th>enum</th>
<th>Buffer</th>
<th>1</th>
<th>#? put</th>
<th>#! overflow</th>
<th>#! out</th>
</tr>
</thead>
</table>

In Equation 6.21 the constants 0 and $n$ are encoded in the application specific constraint script. The structure of canonical checkable properties is described in [13, Section B.3.1 on page 111].

### 6.4.1 Analyzing Checkable Properties

The details of updating and evaluating checkable properties are provided in [13, Section B.2.4 on page 103] and [13, Section B.2.5 on page 105]. In the following paragraphs we indicate how each checkable property is treated by the offline trace analyzer. Table 6.3 summarizes how checkable properties are implemented and analyzed.
**Message precondition.** For every message precondition ‘m(v) assert(expr(v,s))’, a trace database only contains records of failed assertion assert(expr(v,s)). Analysis of a message precondition requires that each failure be reported and the total number of failures for each assertion be enumerated.

**Enumeration constraint.** For every enumeration #z, a trace database contains records of the form Log('r',⋯) for an event #?m or records of the form Log('s',⋯) for an event #!m. Each time an enumeration event for constraint φ occurs, the trace analyzer calls function Evaluate() to check if constraint φ still applies. Analysis of enumeration constraints requires that each failure is reported and that the total number of failures be enumerated. For constraints involving k enumeration terms the evaluation script requires k instance variables n₁,⋯,nₖ to hold the current values of enumerations #z₁,⋯,#zₖ. The update of any term #z forces a reevaluation of the constraint.

**Follows constraint.** For every follows event z₁⇝z₂ associated with constraint φ, a trace database contains records of the form Log('r',⋯) for an event #?m₁ or #?m₂; or records of the form Log('s',⋯) for an event #!m₁ or #!m₂. Each time event z₂ occurs after event z₁ has been recorded, the trace analyzer computes δ=′z₂−''z₁ and calls function Evaluate() to force a reevaluation of constraint φ. Analysis of follows events requires that each failure is reported and that the total number of failures be enumerated. The analyzer accumulates δ and provides a statistical summary of the follows event associated with constraint φ when analysis is complete.

**Timing constraint.** For every timed interval 'z₁ − ''z₂ associated with constraint φ (here we describe the case when z₁ = z₂), a trace database contains records of the form Log('r',⋯) for an event '?m or '''?m; or records of the form Log('s',⋯) for an event '!m or '''!m. Each time event z occurs, the trace analyzer computes δ='z−''z, records the time associated with 'z, and calls function Evaluate() to force a reevaluation of constraint φ. Analysis of timed interval requires that each failure is reported and that the total number of failures be enumerated. The analyzer accumulates δ and provides a statistical summary of the timed interval associated with constraint φ when analysis is complete.
Table 6.3: Summary of checkable property implementation and analysis.

<table>
<thead>
<tr>
<th>Checkable Property</th>
<th>Instrumentation</th>
<th>Monitoring</th>
<th>Analyzer</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>message precondition $m(v)$ \texttt{assert}(\texttt{expr}(v,s))</td>
<td>Use template 1 for $m$.</td>
<td>Evaluate assertion \texttt{expr}(v,s) and log failures.</td>
<td>Count assertion failures.</td>
<td></td>
</tr>
<tr>
<td>enumeration constraint $l \leq #?m_1 - #!m_2 \leq u$</td>
<td>Use template 2 for event $?m_1$, and template 3 for event !$m_2$.</td>
<td>Log events $?m_1$ and !$m_2$.</td>
<td>Count event $?m_1$ ($n_1$), and event !$m_2$ ($n_2$). When either event occurs evaluate using current values of $n_1$ and $n_2$.</td>
<td>Perform the evaluation $l \leq n_1 - n_2 \leq u$. Report failures.</td>
</tr>
<tr>
<td>follows constraint $?m_1 \leadsto !m_2 : n$</td>
<td>Use template 2 for event $?m_1$, and template 3 for event !$m_2$.</td>
<td>Log events $?m_1$ and !$m_2$.</td>
<td>When event !$m_2$ occurs compute $\delta = !m_2 - ?m_1$ then accumulate and evaluate $\delta$.</td>
<td>Perform the evaluation $\delta \leq n$. Report failures when $\delta &gt; n$.</td>
</tr>
<tr>
<td>timing constraint $l \leq t\lbrack?m_1 - !m_2\rbrack \leq u$</td>
<td>Use template 2 for event $?m_1$, and template 3 for event !$m_2$.</td>
<td>Log events $?m_1$ and !$m_2$.</td>
<td>When event $?m_1$ occurs compute $\delta = ?m_1 - !m_2$ then accumulate and evaluate $\delta$.</td>
<td>Perform the evaluation $l \leq \delta \leq u$. Report failures.</td>
</tr>
</tbody>
</table>

\textbf{Templates}\textsuperscript{2}

\[
\begin{align*}
\text{Log}('r', \ldots, ?m, \ldots, & \text{now}, q) ; \\
\text{if (assert(\text{expr}(v,s) == false)} & \\
1 \text{ Log}('p', \phi, \text{false}); \\
\ldots & \\
\text{Log}('c', \ldots, r, q_{\text{next}}) ; \\
\text{Log}('r', \ldots, ?m_1, \ldots, & \text{now}, q) ; \\
2 \ldots & \\
2 \text{ Log}('c', \ldots, r, q_{\text{next}}) ; \\
\text{Log}('r', \ldots, ?m_1, \ldots, & \text{now}, q) ; \\
3 \ldots & \\
3 \text{ Log}('s', \ldots, !m_2, b) ; \\
\ldots & \\
\text{Log}('c', \ldots, r, q_{\text{next}}) ;
\end{align*}
\]

\textsuperscript{2}A description of each type of event log entry is found in Section C.1.
6.5 Limitations of Our Approach

Monitoring does not affect the control flow of an actor, but monitoring activity can affect the performance of an actor; the greater the number of checkable properties to monitor the more intrusive monitoring activity becomes. Only communication events and message preconditions referenced in the definition of a checkable property of a monitored class are monitored and logged. This minimizes the intrusive behaviour of monitoring activity. Enumeration, timing, and follows constraints are evaluated by the offline trace analyzer (see Section 6.4). Message preconditions are evaluated at run-time to avoid unnecessary logging of state and message variables.

The strategy of online monitoring coupled with offline trace analysis is selected to minimize the impact of monitoring on the performance of monitored actors. The nature of the analysis required by certain types of checkable properties suggest that validation strategies which couple online monitoring with online analysis would adversely affect the performance of monitored actors. An important benefit of our present strategy is that the logs produced by an implementation are persistent and subject to reanalysis.

In our approach every message handler associated with a checkable property $\phi$ requires a minimum of two log statements; $\text{Log}(\cdot', \cdot)$ which records timing information at activation, and $\text{Log}(\cdot c', \cdot)$ which records the commitment operator. For a communication of the form $?m$ no additional monitoring is required since $\text{Log}(\cdot r', \cdot)$ satisfies all checkable properties which reference terms of the form $?m, #?m, '/?m,$ and $''?m$. Each transmission event require an additional log statement $\text{Log}(\cdot s', \cdot)$ which satisfies all checkable properties which reference terms of the form $!m, #!m, '!m,$ and $''!m$.

When monitoring is enabled, the length of a run for implementation $E$ is limited by the amount of available memory at each participating node. For our validation approach and case studies like those presented in Chapter 7, this limitation is not encountered.

The effect of monitoring on running time. We conducted the following experiments to determine the effect of monitoring on the response time of event $!f \sim ?g$ in application DABP (see Section 7.2). We performed ten runs of application DABP for each of the following treatments:
treatment \( X_{\text{none}} \) was performed with monitoring disabled, treatment \( X_{\phi_1, \phi_3} \) was performed with only checkable properties \( \phi_1 \) and \( \phi_3 \) enabled (assertions), treatment \( X_{\phi_2} \) was performed with only checkable property \( \phi_2 \) enabled (a follows constraint), and treatment \( X_{\phi_4} \) was performed with only checkable property \( \phi_4 \) enabled (a timing constraint).

For each run we measured the average response time of 102,400 control messages \((g())\) from the perspective of the ABP sender. We denote this response time by the interval \( f \xrightarrow{c} g \). These average response times are listed in the column labelled Mean in Table 6.4.

<table>
<thead>
<tr>
<th>Treatment ((i))</th>
<th>Mean (X_i)</th>
<th>Variance (s_i^2)</th>
<th>(c)</th>
<th>(t)</th>
<th>(df)</th>
<th>(P) (^3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitoring disabled</td>
<td>43.8</td>
<td>0.1781</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monitor (\phi_1, \phi_3)</td>
<td>47.6</td>
<td>0.4886</td>
<td>0</td>
<td>14.717</td>
<td>18</td>
<td>(P(t) \leq 0.005)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>6.971</td>
<td>18</td>
<td>(P(t) \leq 0.005)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3.983</td>
<td>18</td>
<td>(P(t) \leq 0.005)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monitor (\phi_2)</td>
<td>47.9</td>
<td>0.0999</td>
<td>0</td>
<td>24.600</td>
<td>18</td>
<td>(P(t) \leq 0.005)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12.595</td>
<td>18</td>
<td>(P(t) \leq 0.005)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>6.597</td>
<td>18</td>
<td>(P(t) \leq 0.005)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monitor (\phi_4)</td>
<td>46.2</td>
<td>0.3994</td>
<td>0</td>
<td>9.985</td>
<td>18</td>
<td>(P(t) \leq 0.005)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>5.826</td>
<td>18</td>
<td>(P(t) \leq 0.005)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.665</td>
<td>18</td>
<td>(0.10 \leq P(t) \leq 0.05)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data Sets

\(X_{\text{none}} = \{44,44,44,44,44,44,44,44,44\}, \ n = 10\)
\(X_{\phi_1, \phi_3} = \{47,48,47,47,48,47,47,48,48,49\}, \ n = 10\)
\(X_{\phi_2} = \{48,47,48,48,48,48,48,48,48,48\}, \ n = 10\)
\(X_{\phi_4} = \{46,46,46,46,46,46,46,46,46,48\}, \ n = 10\)

Generalized \(t\)-statistic

\[ t = \frac{|X_i - X_j| - c}{\sqrt{s_i^2 + s_j^2}} \]

where \(X_i = X_{\text{none}}\), and \(X_j \in \{X_{\phi_1, \phi_3}, X_{\phi_2}, X_{\phi_4}\}\)

\[ H_0 : \mu_i - \mu_j \leq c, \ H_A : \mu_i - \mu_j > c \]

The null hypothesis for all experiments is \(H_0 : \mu_i - \mu_j \leq c\), i.e. the difference between the average response time of control messages when checkable properties are disabled (\(X_{\text{none}}\)) and the

\(^3\)In this table \(c\) is the constant in the generalized \(t\)-statistic; \(t\) is the generalized \(t\)-statistic; \(df\) is the degrees of freedom; and \(P\) is the probability (1-tailed) of obtaining a \(t\)-statistic greater than or equal to the value tabulated in column \(t\).
average response time of control messages with selected checkable properties enabled (\(\bar{X}_j\) for \(X_j \in \{X_{\phi_1}, X_{\phi_3}, X_{\phi_2}, X_{\phi_4}\}\)) is less than or equal to some constant \(c\). The \(t\) statistics comparing \(\bar{X}_{none}\) with each \(\bar{X}_j\) for several values of \(c\) are summarized in Table 6.4. In all cases we reject \(H_0\) and state there is a significant difference in the average response times for all paired treatments \(\bar{X}_{none}\) with \(\bar{X}_j\); we accept the alternate hypothesis \(H_A: \mu_i - \mu_j > c\).

Instrumenting application DABP has the following effects on running time:

- monitoring properties \(\phi_1\) and \(\phi_3\) (assertions) adds approximately 3 \(\mu s\) to the mean response time \(f \leadsto g\), or about 2 \(\mu s\) of processing for each monitored message at the sending and receiving nodes,

- monitoring property \(\phi_2\) (a follows constraint) adds approximately 3 \(\mu s\) to the mean response time \(f \leadsto g\), and

- monitoring property \(\phi_4\) (a timing constraint) adds approximately 2 \(\mu s\) to the mean response time \(f \leadsto g\).

**Interpretation of timing statistics.** The trace analyzer provides statistical summaries for every monitored follows and timing constraint. Figure 6.10 is a scatterplot of the running time of application DABP (see Section 7.2) versus the round-trip times for event \(?nf \leadsto ?g\), extracted from the data set for Trial 1 of application DABP.

For this two second interval the mean round-trip time is reported as 203.7 ± 409.7 \(\mu s\) \((n = 9131)\). All follows and timing constraints examined for this dissertation experience high jitter. This high jitter is in part due to the following factors:

- Timeouts (COOL timers)—For application DABP we record 33 timeouts over the two second interval illustrated in Figure 6.10. All timeouts are attributable to the acknowledgement timer of the ABP sender. Since the period of the acknowledgement timer is 4 ms, many of the high-latency round-trip values are attributable to this timer.

- Context switches and suspensions—COOL executables are subject to host operating system context switches and suspensions. For application DABP this affects both the sending and
receiving node. If the host operating system provides a quantum of 20 ms, application DABP could experience up to 200 context switches (or suspensions) of varying duration for the two second interval illustrated in Figure 6.10.

![Figure 6.10: Running time versus round-trip time in application DABP.](image)

### 6.6 Summary

Formal approaches to the analysis of concurrent systems generally involve two notations: a notation for describing structural changes in a system (e.g., process algebras, process calculi, and labelled transition systems), and a notation for expressing system properties (e.g., modal logic, and temporal logic). In our Timed Actor model, analysis is based on

- an *asynchronous actor algebra* described in Chapter 3, and
- a notation for expressing *checkable properties*.

Checkable properties include *message preconditions*, an assertion which is evaluated each time a message handler is activated; *enumeration constraints*, which collect information on the frequency
of selected *communication events* (message reception and transmission); *timing constraints*, which measure timed intervals between two communication events; and *follows constraints*, which are used to determine if two causally related communication events occur within a specified period of time.

Checkable properties can be defined for any COOL actor class \( C \) to describe the *expected behaviour* of an actor \( a \in C \). When the COOL compiler encounters the definition of a checkable property in a COOL specification it instruments the implementation to enable monitoring of the specified checkable property (Section 6.3). When the implementation is executed the VM and online monitor log event traces to an in-memory database. When execution is complete the VM prepares a log of the *observed behaviour* of monitored actors (Section 6.3.1). Logs created by the VM are analyzed by an offline trace analyzer which reports when the observed behaviour of each monitored actor differs from the expected behaviour defined in the actor class specification (Section 6.4).
Chapter 7

Case Studies

In Chapter 5 we presented the syntax and informal semantics of COOL, a timed actor language for specifying the interfaces, behaviour, and coordination of actors which implement distributed applications. In Section 1.1.2 we briefly describe how specifications are translated into executable applications. COOL specifications may include definitions of checkable properties (see Sections 6.1 and 6.2), which allow us to describe the expected behaviour of selected actors. Using the techniques described in Sections 6.3–6.4 we can compare the observed behaviour of actors during execution with their defined expected behaviour.

In this chapter we describe our experience with designing, implementing, and analyzing selected distributed protocols. In the cases presented, we emphasize the more important features of the COOL specification language; timing, coordination, commitment strategy (the use of abort), and validation using the monitoring approach described in Chapter 6. The case studies described in Sections 7.2–7.3 are presented using a similar format. Initially, we describe a configuration for each case study; including the input sets, and transition relations of each primary actor class. Next, we illustrate the normal exchange of messages in the configuration using message sequence charts [7]. We also illustrate the effect of lost control messages on the protocol with the aid of message sequence charts. Finally, we analyze the monitoring results collected for each case (Section 6.4).

In Section 7.2 we describe application DABP, an implementation of the alternating bit protocol (ABP). A common requirement of many distributed applications is that the communications
medium be reliable and order preserving. In all COOL applications internodal communication is unreliable and not order preserving. ABP is one protocol for achieving reliable and order preserving communications. Our implementation of ABP demonstrates the use of the abort operator.

In Section 7.3 we describe application DLOCK, a simple simulation for validating a distributed locking protocol. The distributed locks tested in this simulation can be used to protect distributed resources in environments where the number of users $n$ greatly outnumber the arity of the resource $m$. In lossy environments where there is a high degree of competition for a resource, we expect control messages to be lost, duplicated, or corrupted. The distributed locking protocol employs a lock coordinator to access a distributed lock. The lock coordinator uses a bounded retransmission strategy to make the control channel between a lock user and the distributed lock reliable. Our implementation of DLOCK utilizes several enumerations to validate the operation of the distributed lock.

7.1 Programming Environment and Some Additional Notation

Figure 1.2 (see Section 1.1.2) illustrates the key steps in producing an executable application from a COOL specification. The case studies described in this chapter are implemented using the cool2C translator described in [14, 13]. The implementations generated by cool2C are compiled using the native C/C++ compiler for the environments listed in Table 7.1. A compiled implementation is bound to a C/C++ version of the run-time kernel [15] to produce an executable application.

<table>
<thead>
<tr>
<th>Hostname</th>
<th>alpha</th>
<th>beta</th>
<th>gamma</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>733 MHz Pentium III</td>
<td>300 MHz Pentium II</td>
<td>1 GHZ Pentium IV</td>
</tr>
<tr>
<td>Operating System</td>
<td>Windows XP</td>
<td>Red Hat Linux 7.2</td>
<td>Red Hat Linux 9.0</td>
</tr>
<tr>
<td>Memory</td>
<td>512M</td>
<td>128M</td>
<td>512M</td>
</tr>
<tr>
<td>Compiler¹</td>
<td>cl 13.00</td>
<td>g++ 2.96</td>
<td>g++ 3.2.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application DABP</td>
</tr>
<tr>
<td>Application DLOCK</td>
</tr>
</tbody>
</table>
Denoting special enumerations. Trace analysis provides a summary of the observed behaviour of selected actors as well as the enumeration of certain special events. Our offline trace analyzer ((see 13, Section B.2 on page 100)) reports

- the frequency of rollbacks (generated using `abort`) occurring in message handler \( m \), denoted \#\text{rollback}(m);
- the frequency with which message \( m \) is dropped (see Section 5.1.5 on page 111), denoted \#\text{drop}(m);
- the frequency with which error \( e \) (see Section 4.4.3) occurs, denoted \#\text{error}(e); and
- the frequency with which message preconditions referencing message handler \( m \) fail, denoted \#\text{assert}_f(m).

### 7.2 Application DABP—Alternating Bit Protocol

Alternating bit protocol (ABP) demonstrates how a single unreliable communications channel can be made reliable. ABP is used to move a sequence of data frames from a sender to a receiver. Each data frame received by the receiver is acknowledged by a return control frame to the originating sender. All data and control frames utilize a binary sequence number to preserve the delivery order of data frames. ABP can handle lost, corrupt and duplicate data and control frames [53].

ABP is a distributed protocol, the sender and receiver maintain private instance variables to record the sequence number of the last data frame transmitted (Sender) or received (Receiver). In a correct ABP implementation a sender will not deadlock due to lost data or control frames. To avoid deadlock, the sender employs an acknowledgement timer to guard against loss of data and control frames. If the duration of an acknowledgement timer is set correctly, the receiver will not be overrun. An implementation of the ABP sender and receiver described here is given in Listings D.4 and D.5 of Appendix D.1.

In this case study we are interested in answering the following questions:

\(^1\)cl is the Microsoft® 32-bit C++ optimizing compiler, and g++ is the GNU C++ optimizing compiler.
• Does the implementation preserve the order of data frames?

• Does the implementation handle lost and duplicate data and control frames correctly?

• Can we estimate the throughput of data frames from the perspective of receiver r?

• Can we estimate the end-to-end delay of control frames from the perspective of sender s?

Section 7.2.1 describes the configuration used to study ABP. An analysis of our ABP implementation is provided in Section 7.2.2.

### 7.2.1 A Configuration for Studying Alternating Bit Protocol

The components of an ABP configuration are summarized in Table 7.2. In this configuration

\[
(s \uparrow t_{ack}) \mid u_s
\]

\[
r \mid u_r
\]

Equation 7.1 defines the sending node, and Equation 7.2 defines the receiving node. Actors s and r provide an ABP service, while actors us and ur simulate service access points provided by the network layer. Timer \( t_{ack} \) is an acknowledgement timer local to actor s.

Table 7.2: Components and input sets for application DABP.

<table>
<thead>
<tr>
<th>Components</th>
<th>Input Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u_s \in User )</td>
<td>( \Sigma_{us} \supseteq {sapa()} )</td>
</tr>
<tr>
<td>( s \in Sender )</td>
<td>( \Sigma_s = {init(actor), nf(int), g(int), noack(int,int), stop()} )</td>
</tr>
<tr>
<td>( t_{ack} \in Timer )</td>
<td>( \Sigma_{us} \supseteq {sapa()} )</td>
</tr>
<tr>
<td>( u_r \in User )</td>
<td>( \Sigma_{us} = {init(actor), nf(int), g(int), noack(int,int), stop()} )</td>
</tr>
<tr>
<td>( r \in Receiver )</td>
<td>( \Sigma_r = {init(actor), f(int,int), stop()} )</td>
</tr>
</tbody>
</table>

Figure 7.1 illustrates how our ABP implementation assigns actors to nodes. In ABP, data frames move from the sender to the receiver, while control frames move from the receiver to the sender. In our configuration actor \( u_s \) simulates the network layer on the node where sender \( s \) resides. The
sender requests and receives data frames through \( u_s \). Actor \( u_r \) simulates the network layer on the node where receiver \( r \) resides. As data frames \( (x) \) are received by \( r \), they are passed to the network layer through \( u_r \).

![Diagram](image)

**Figure 7.1:** Application DABP—assigning actors to nodes.

### 7.2.1.1 An Implementation of ABP

The behaviour of \( s \in \text{Sender} \) is expressed as a transition relation in Table 7.3. The sender \( s \) initiates the protocol by requesting a data frame from the network layer \( (\text{sap}(\_)) \). When a data frame arrives \( (\text{nf}(x)) \) the sender attaches a sequence number \( (\text{nfts}) \) to the frame, outputs the frame to the receiver \( (\text{f}(\text{nfts}, x)) \), starts acknowledgement timer \( t_{\text{ack}} \); then commits, waiting for a reply from the receiver \( (\text{g}(\text{seq})) \).

<table>
<thead>
<tr>
<th>( q_{\text{current}} )</th>
<th>( \Sigma_{\text{Sender}} )</th>
<th>Condition</th>
<th>( q_{\text{next}} )</th>
<th>( K^+ )</th>
<th>( J^+ / J^- )</th>
<th>( X^+ )</th>
<th>( R )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Start</strong></td>
<td>init(( x ))</td>
<td></td>
<td>Tx</td>
<td>( { } )</td>
<td>( { } )</td>
<td>( [u.\text{sap}(_)] )</td>
<td>becomes Tx</td>
</tr>
<tr>
<td>Tx</td>
<td>( \text{nf}(x) )</td>
<td>( \text{nfts} = \text{seq} )</td>
<td>Tx</td>
<td>( { } )</td>
<td>( {+t_{\text{ack}}} )</td>
<td>( [r.\text{f}(\text{nfts}, x)] )</td>
<td>commit</td>
</tr>
<tr>
<td>Tx</td>
<td>( \text{g}(\text{seq}) )</td>
<td>( \text{nfts} \neq \text{seq} )</td>
<td>Tx</td>
<td>( { } )</td>
<td>( {-t_{\text{ack}}} )</td>
<td>( [u.\text{sap}(_)] )</td>
<td>commit</td>
</tr>
<tr>
<td>Tx</td>
<td>( \text{noack}(\text{seq}, x) )</td>
<td></td>
<td>Tx</td>
<td>( { } )</td>
<td>( { } )</td>
<td>( [_] )</td>
<td>commit</td>
</tr>
<tr>
<td><strong>stop()</strong></td>
<td></td>
<td></td>
<td>0</td>
<td>( { } )</td>
<td>( { } )</td>
<td>( [_] )</td>
<td>terminate</td>
</tr>
</tbody>
</table>
The sender is reactivated by one of the following events:

- The receiver responds with a positive acknowledgement (g(seq) with nfts = seq). In this case the sender discards timer t_{ack}, increments frame counter nfts, then requests the next data frame from the network layer (sapa()) (see Figure 7.2(a)).

- The receiver responds with a negative acknowledgment (g(seq) with nfts $\neq$ seq). In this case the sender takes no action, treating the negative acknowledgement as a lost control frame. This strategy results in the eventual timeout of t_{ack} and the retransmission of the last data frame (see Figure 7.2(b)).

- The acknowledgement timer expires (noack(seq,x)). This indicates that either the last data frame was lost, the last control frame was lost, or the last control frame was a duplicate. In all cases the sender retransmits the last data frame (see Figure 7.2(b)).

The behaviour of $r \in \text{Receiver}$ is expressed as a transition relation in Table 7.4. The receiver waits for an incoming data frame (f(seq,x)). If the frame does not contain the expected sequence number ($fe \neq seq$) the receiver sends a negative acknowledgement to the sender containing the sequence number of the last frame received (g(dec(fe))). If the frame contains the correct sequence number ($fe = seq$), the receiver increments frame counter $fe$, checks the validity of the data (checkData(x)), delivers the data frame to the network layer (sapb(x)), sends a positive acknowledgement to the sender (g(seq)), then resumes waiting for the next incoming data frame.

<table>
<thead>
<tr>
<th>$q_{current}$</th>
<th>$\Sigma_{\text{Receiver}}$</th>
<th>Condition</th>
<th>$q_{next}$</th>
<th>$K^+$</th>
<th>$X^+$</th>
<th>$R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>init(x)</td>
<td></td>
<td>Rx</td>
<td>$\emptyset$</td>
<td></td>
<td>becomes Rx</td>
</tr>
<tr>
<td>Rx</td>
<td>f(seq,x)</td>
<td>checkData(x) = false</td>
<td>Rx</td>
<td>$\emptyset$</td>
<td></td>
<td>abort</td>
</tr>
<tr>
<td>Rx</td>
<td>f(seq,x)</td>
<td>checkData(x) = true</td>
<td>Rx</td>
<td>$\emptyset$</td>
<td>[u.sapb(x) $\cdot$ s.g(seq)]</td>
<td>commit</td>
</tr>
<tr>
<td>Rx</td>
<td>f(seq,x)</td>
<td>$fe \neq seq$</td>
<td>Rx</td>
<td>$\emptyset$</td>
<td>[s.g(dec(fe))]</td>
<td>commit</td>
</tr>
<tr>
<td>*</td>
<td>stop()</td>
<td></td>
<td>0</td>
<td>$\emptyset$</td>
<td></td>
<td>terminate</td>
</tr>
</tbody>
</table>

When function checkData() (Listing D.6) detects a corrupt datum, the receiver abandons the
processing of message \( f() \) using the \textbf{abort} operation. Our implementation of function \texttt{checkData()} randomly generates corrupt data to test whether rollbacks are being handled correctly by the runtime system.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure.png}
\caption{Message sequence charts illustrating the interaction between a sender \( s \in \text{Sender} \) and a receiver \( r \in \text{Receiver} \) in ABP.}
\end{figure}

\textbf{Frame counters.} Figure 7.3 illustrates when the frame counters, \( nfts \) and \( fe \), are updated by the sender and receiver in our implementation of ABP. \( nfts \) provides the sequence number for data frames, while \( fe \) indicates what frame the receiver is expecting.
Figure 7.3: Alternating bit protocol—changes in frame counters nfts (Sender) and fe (Receiver).

7.2.2 An Analysis of Application DABP

To examine the characteristics of DABP, we designed a simulation with a single sender $s$ (Node 1) and a single receiver $r$ (Node 2). During each of five trials the sender transmitted 102,400 data frames to the receiver. At the end of the simulation actor $u_r$ reported the number of out of order frames reaching the network layer.

**Sequence control.** Every implementation of ABP must enforce the *sequence invariant*

$$nf(x[i]) = sapb(x[i]), \quad \forall i, \ 1 \leq i \leq n$$

which reads “for a data stream of $n$ frames the $i$-th frame produced by actor $u_s$ ($nf()$) must be equal to the $i$-th data frame consumed by actor $u_r$ ($sapb()$)”. Rows 1 and 2 in Table 7.5 summarize the principal result of our study of ABP. In the five reported trials our implementation successfully transferred all data frames while preserving their order. This transfer was accomplished in the presence of rollback, dropped data frames, and dropped control frames.

**Handling lost frames.** We set the *drop probability* for data frames ($f(seq,x)$) and control frames ($g(seq)$) to 0.001, to test whether our implementation of ABP correctly handles situations involving
lost data and control frames. The expected outcome of dropping data and control frames is the
generation of timeouts by timer $t_{ack}$. Rows 3, 4, 5, 6, and 8 in Table 7.5 indicate that for this
implementation of ABP, the frequency of timeouts by timer $t_{ack}$ ($\#?noack$) is related to the following
events

\[
\#?noack = \#rollback(f) + \#drop(f) + \#drop(g) + \#assert_f(g)
\]  \hspace{1cm} (7.4)

Any occurrence of these four events result in a timeout of timer $t_{ack}$. This result indicates that our
implementation of ABP can handle lost frames, and rollback while preserving the order of data
frames.

**Handling duplicate frames.** Figure 7.2(b) illustrates that the loss of control frames leads directly
to the generation of duplicate data frames by sender $s$. We employ a message precondition

\[
\phi_3 \triangleq f(seq,x) \text{ assert}(fe == seq)
\]

at the receiver to monitor the occurrence of duplicate data frames. This constraint is asserted when
the sequence number of a data frame ($seq$) does not match the value of frame counter $fe$. Row 7 in
Table 7.5 shows that this message precondition is effective in monitoring duplicate data frames.

Figure 7.4 illustrates one way in which duplicate control frames are generated by ABP. The
duration of the acknowledgement timer $t_{ack}$ is important in determining the performance of ABP. If
the duration of $t_{ack}$ is too long, delayed data and control frames have additional time to reach their
destination, but overall performance of the protocol can suffer. If the duration of $t_{ack}$ is too short,
$t_{ack}$ will timeout prematurely, and duplicate control frames will be generated by the receiver. In our
implementation of ABP duplicate control frames are ignored by the sender.

We employ a message precondition

\[
\phi_1 \triangleq g(seq) \text{ assert}(nfts == seq)
\]

at the sender to monitor the occurrence of duplicate control frames. This constraint is asserted when
the sequence number of a control frame ($seq$) does not match the value of the frame counter $nfts$. 

Row 8 in Table 7.5 shows that this message precondition is effective in monitoring duplicate control frames.

![Message Sequence Chart](image)

Figure 7.4: A message sequence chart illustrating the effect of incorrectly setting the duration of the acknowledgement timer in ABP.

**Throughput of data frames.** The specification of class `Receiver` in Listing D.5 contains the `timing constraint`

\[ \phi_4 \triangleq 0 \leq \text{'}f - \text{"}f \text{'} \leq 300 \text{ usec} \]

which is used to measure the interarrival times of data frames from the perspective of receiver \( r \). Our analysis, summarized in Row 9 of Table 7.5, indicates that the mean interarrival time of data frames varies between 178.5–180.7 \( \mu s \) with a jitter of 39.5–40.0 \( \mu s \). This results in an estimated throughput of 5534–5602 frames per second with a 22% jitter.
End-to-end delay of control frames. The specification of class Sender in Listing D.4 contains the follows constraint

$$\phi_2 \triangleq ?n.f \sim ?g : 300 \texttt{usec}$$

which is used to measure the response time of frames in an ABP configuration. Our analysis, summarized in Row 10 of Table 7.5, indicates that the mean response time varies between 165.6–167.5 $\mu s$ with a jitter of 38.3–40.0 $\mu s$. Since local clocks are not synchronized, the best estimate of the end-to-end delay of control frames is 82.8–83.8 $\mu s$, one-half of the measured response times.
Table 7.5: Analysis results for application DABP.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Class</th>
<th>Trial 1</th>
<th>Trial 2</th>
<th>Trial 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Number of frames successfully transmitted</td>
<td>Users</td>
<td>102400</td>
<td>102400</td>
<td>102400</td>
</tr>
<tr>
<td>(2) Number of out of sequence data frames</td>
<td>Users</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(3) Number of rollbacks, (#\text{rollback}(f))</td>
<td>Receiver</td>
<td>98</td>
<td>104</td>
<td>101</td>
</tr>
<tr>
<td>(4) (#\text{drop}(f), p = 0.001)</td>
<td>Receiver</td>
<td>108</td>
<td>102</td>
<td>106</td>
</tr>
<tr>
<td>(5) (#\text{drop}(g), p = 0.001)</td>
<td>Sender</td>
<td>109</td>
<td>107</td>
<td>107</td>
</tr>
<tr>
<td>(6) Number of timeouts, (#\text{nack})</td>
<td>Sender</td>
<td>409</td>
<td>422</td>
<td>412</td>
</tr>
<tr>
<td>(7) (#\text{assert}_f(f, \text{seq}, x) \ \	ext{assert}(\text{fe} == \text{seq}))</td>
<td>Receiver</td>
<td>203</td>
<td>216</td>
<td>205</td>
</tr>
<tr>
<td>(8) (#\text{assert}_r(g, \text{seq}) \ \	ext{assert}(\text{nfts} == \text{seq}))</td>
<td>Sender</td>
<td>94</td>
<td>109</td>
<td>98</td>
</tr>
<tr>
<td>(9) Mean interarrival time (\delta &lt; 1000)</td>
<td>Receiver</td>
<td>180.7 ± 39.5(\mu s)</td>
<td>178.5 ± 40.0(\mu s)</td>
<td>180.4 ± 40.5(\mu s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(n = 100922)</td>
<td>(n = 100648)</td>
<td>(n = 100954)</td>
</tr>
<tr>
<td>(10) Mean response time (\delta &lt; 1000)</td>
<td>Sender</td>
<td>167.5 ± 38.3(\mu s)</td>
<td>165.6 ± 40.0(\mu s)</td>
<td>167.5 ± 40.5(\mu s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(n = 101073)</td>
<td>(n = 100830)</td>
<td>(n = 101133)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric</th>
<th>Class</th>
<th>Trial 4</th>
<th>Trial 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Number of frames successfully transmitted</td>
<td>Users</td>
<td>102400</td>
<td>102400</td>
</tr>
<tr>
<td>(2) Number of out of sequence data frames</td>
<td>Users</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(3) Number of rollbacks, (#\text{rollback}(f))</td>
<td>Receiver</td>
<td>99</td>
<td>95</td>
</tr>
<tr>
<td>(4) (#\text{drop}(f), p = 0.001)</td>
<td>Receiver</td>
<td>108</td>
<td>111</td>
</tr>
<tr>
<td>(5) (#\text{drop}(g), p = 0.001)</td>
<td>Sender</td>
<td>107</td>
<td>108</td>
</tr>
<tr>
<td>(6) Number of timeouts, (#\text{nack})</td>
<td>Sender</td>
<td>416</td>
<td>427</td>
</tr>
<tr>
<td>(7) (#\text{assert}_f(f, \text{seq}, x) \ \	ext{assert}(\text{fe} == \text{seq}))</td>
<td>Receiver</td>
<td>209</td>
<td>221</td>
</tr>
<tr>
<td>(8) (#\text{assert}_r(g, \text{seq}) \ \	ext{assert}(\text{nfts} == \text{seq}))</td>
<td>Sender</td>
<td>102</td>
<td>113</td>
</tr>
<tr>
<td>(9) Mean interarrival time (\delta &lt; 1000)</td>
<td>Receiver</td>
<td>179.6 ± 39.7(\mu s)</td>
<td>180.3 ± 40.1(\mu s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(n = 100721)</td>
<td>(n = 100861)</td>
</tr>
<tr>
<td>(10) Mean response time (\delta &lt; 1000)</td>
<td>Sender</td>
<td>166.8 ± 39.8(\mu s)</td>
<td>167.3 ± 38.9(\mu s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(n = 100901)</td>
<td>(n = 101035)</td>
</tr>
</tbody>
</table>
7.3 Application DLOCK—A Distributed Lock Protocol

In a distributed system it is common for service users to acquire tokens before accessing shared services. Once acquired tokens can age, and losing a token through aging may leave a service user in an undefined state. This should motivate service users to divide their work into discrete steps, acquiring and releasing tokens as necessary to avoid costly recovery procedures.

A distributed lock is a simple mechanism for implementing tokens. In a messaging environment locks are acquired by sending lock requests to a lock service. When a lock is granted the lock service associates the identity of the requester with the lock.

In an asynchronous environment with lossy communications, the probability of lost requests and acknowledgements is high and a coordination object may be required to achieve a reliable exchange of control messages between the user and lock service.

In this section we describe a simulation of a distributed lock protocol which has the following characteristics:

- Service users \( u \in \text{User} \) communicate with the distributed lock \( d \in \text{DLock} \) indirectly through a local lock coordinator \( l \in \text{Locker} \). Once a lock request is issued to a coordinator \( l \), the service user \( u \) may return to other tasks.

- The lock coordinator and distributed lock follow a locking protocol. This protocol utilizes a bounded retransmission strategy to make communications between the lock and lock coordinator reliable. Once granted, a distributed lock can be held for a fixed period of time, referred to as the quantum.

- Once the lock is acquired a service user has exclusive access to a service \( s \in \text{Service} \) protected by lock \( d \). In this simulation the service user \( u \) utilizes a proxy \( p \in \text{Proxy} \) to manage all access to service \( s \). In this discussion we are not concerned with either the nature of service \( s \), or the interactions between the proxy \( p \) and the service \( s \).

- The proxy \( p \) indicates to the service user \( u \) when it should issue an unlock request. Figure 7.5 illustrates the normal operation of this simulation.
• If the lock holder exceeds its quantum, distributed lock $d$ discards service $s$ and reclaims the lock. Figure 7.6 illustrates this situation.

In this simulation we allow $n$ service users and one lock service. We assess the correctness of the locking protocol by employing the *binary lock invariant* (see Section 6.2 on page 134). An implementation of the distributed lock simulation described in this section is given in Listings D.7–D.10 of Appendix D.1.

![Distributed Lock Simulation — Normal Operation](image)

Figure 7.5: Application DLOCK, message sequence chart illustrating a successful service access.

In this case study we are interested in answering the following questions:

• Does the distributed lock $d \in \text{DLock}$ obey the *binary lock invariant*?
• What failure modes are observed by a user $u \in \text{User}$?

• Do the distributed lock $d \in \text{DLock}$ and the lock coordinator $l \in \text{Locker}$ correctly handle lost and duplicate control messages?

Section 7.3.1 describes the configuration used to study the distributed lock protocol. An analysis of the distributed lock protocol is provided in Section 7.3.2.

![Figure 7.6: Application DLOCK, message sequence chart illustrating an unsuccessful service access.](image)

### 7.3.1 A Configuration for a Distributed Lock Protocol

The components of the distributed lock simulation are summarized in Table 7.6. In this configuration,
Equation 7.5 defines a lock service, and Equation 7.6 defines multiple service users. In the lock service, the distributed lock $d$ controls access to service $s$. The service user holding lock $d$ has exclusive access to service $s$ through proxy $p$. The timer $t_{qt}$ implements the lock quantum. In the user service, $t_{lock}$ controls the retransmission characteristics of lock coordinator $l$.

Table 7.6: Components and input sets for the DLOCK simulation.

<table>
<thead>
<tr>
<th>Components</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$u \in$ User</td>
<td>$d \in$ DLock</td>
<td>$l \in$ Locker</td>
</tr>
<tr>
<td>$p \in$ Proxy</td>
<td>$s \in$ Service</td>
<td>$m \in$ Main</td>
</tr>
<tr>
<td>$t_{req} \in$ Timer</td>
<td>$t_{qt}, t_{lock} \in$ Timer</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma_u = { \text{init}(\text{actor}, \text{int}, \text{int}), \text{nextAttempt}(), \text{locked}(\text{actor}), \text{complete}(), \text{unlocked}(), \text{expired}(), \text{apperror}(\text{int}), \text{stop}() }$</td>
</tr>
<tr>
<td>$\Sigma_d = { \text{init}(\text{int}), \text{lock}(\text{actor}), \text{unlock}(), \text{noquantum}(), \text{stop}() }$</td>
</tr>
<tr>
<td>$\Sigma_t = { \text{init}(\text{int}, \text{int}), \text{lock}(\text{actor}, \text{actor}), \text{locked}(\text{actor}), \text{locked}_r(\text{actor}), \text{reject}(), \text{unlock}(\text{actor}), \text{unlocked}(), \text{unlocked}_r(), \text{noresponse}(\text{int}), \text{stop}() }$</td>
</tr>
<tr>
<td>$\Sigma_p \subseteq { \text{start}(\text{actor}), \text{end}(\text{actor}), \text{reset}(), \text{stop}() }$</td>
</tr>
<tr>
<td>$\Sigma_i \subseteq { \text{init}(\text{actor}), \text{stop}() }$</td>
</tr>
</tbody>
</table>

Figure 7.7 illustrates how actors are assigned to nodes in the distributed lock simulation.
Figure 7.7: A configuration for testing a distributed lock protocol.

Since many users compete for lock \( d \), there is a high probability that requests submitted by coordinator \( l_i \) for the lock will be rejected by \( d \). In such situations we employ a \textit{bounded retransmission strategy} to acquire the lock. In a bounded retransmission strategy the requesting actor repeats a service request up to \( n_1 \) times, waiting \( n_2 \) time units between each request. The requesting actor will either acquire the service or fail within \( n_1 \times n_2 \) time units. This insures a reliable exchange of control messages between the lock and lock coordinator. Once user \( u_i \) acquires lock \( d \) the proxy \( p_i \) can access service \( s \).

7.3.1.1 A Distributed Lock Protocol

Class User—Lock User (Listing D.9). User \( u \) initializes the simulation by establishing the timing characteristics of lock coordinator \( l \) using control message \( \text{init}(n_1, n_2) \), where \( n_1 \) sets the duration of \( t_{\text{lock}} \) and \( n_2 \) sets the number of attempts to secure distributed lock \( d \). The transition relation for lock
users is summarized in Table 7.7.

Table 7.7: Application DLOCK, transition relations for actor \( u \in \text{User} \).

<table>
<thead>
<tr>
<th>Condition</th>
<th>( q_{current} )</th>
<th>( \sum_{User} )</th>
<th>( q_{next} )</th>
<th>( K^+ )</th>
<th>( J^+ / J^- )</th>
<th>( X^+ )</th>
<th>( R )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{init}(d,n_1,n_2) )</td>
<td>Start</td>
<td>Run { p, n }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>Run { }</td>
</tr>
<tr>
<td>( \text{nextAttempt()} )</td>
<td>Run</td>
<td>Run { }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>Run { }</td>
</tr>
<tr>
<td>( C_1 = \text{false} )</td>
<td>Run</td>
<td>Run { }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>Run { }</td>
</tr>
<tr>
<td>( C_1 = \text{true} )</td>
<td>Run</td>
<td>Run { }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>Run { }</td>
</tr>
<tr>
<td>( \text{lock}(s) )</td>
<td>Run</td>
<td>Run { }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>Run { }</td>
</tr>
<tr>
<td>( \text{complete()} )</td>
<td>Run</td>
<td>Run { }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>Run { }</td>
</tr>
<tr>
<td>( \text{unlocked()} )</td>
<td>Run</td>
<td>Run { }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>Run { }</td>
</tr>
<tr>
<td>( \text{expired()} )</td>
<td>Run</td>
<td>Run { }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>Run { }</td>
</tr>
<tr>
<td>( \text{apperror}(x) )</td>
<td>*</td>
<td>Run { }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>Run { }</td>
</tr>
<tr>
<td>( \text{stop()} )</td>
<td>*</td>
<td>Run { }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>{ +t_{req} }</td>
<td>Run { }</td>
</tr>
</tbody>
</table>

User \( u \) requests the lock using control message \( \text{lock}(d,p) \) where \( d \) is the identity of the distributed lock, and \( p \) is the identity of the user’s proxy. From the perspective of user \( u \) there are two outcomes to a \( \text{lock}() \) request:

1. The lock coordinator \( l \) can respond with message \( \text{locked}(s) \) indicating that \( u \) has acquired the lock and has exclusive use of service \( s \). Access to service \( s \) is initiated by sending control message \( \text{start}(s) \) to proxy \( p \). Two events can occur during this access:

   - The proxy \( p \) can indicate a successful access by sending message \( \text{complete()} \) to \( u \). In this case user \( u \) sends an \( \text{unlock}(d) \) request to lock coordinator \( l \) (see Figure 7.5).
   - The lock coordinator \( l \) can indicate an unsuccessful access by sending message \( \text{expired()} \) to \( u \) (proxy \( p \) has exceeded its quantum for service \( s \)). In this case \( d \) discards service \( s \), then reclaims the lock (see Figure 7.6).

2. The lock coordinator \( l \) can respond with message \( \text{apperror}(1) \) indicating that \( l \) was unable to acquire lock within \( n_1 \times n_2 \) time units. In this case service user \( u \) waits for timeout message \( \text{nextAttempt()} \) before making another \( \text{lock}(d,s) \) request.

\(^2\)For class User condition \( C_1 = (n_{Sessions} \geq m_{Sessions}) \) is evaluated when busy = false.
Class DLock—Distributed Lock (Listing D.7). A communicating finite state machine illustrating the behaviour of $d \in \text{DLock}$ is shown in Figure 7.8. The transition relation for this CFSM is summarized in Table 7.8. Actors belonging to class DLock, implement distributed binary locks. Class DLock is designed to interact with a lock coordinator $l \in \text{Locker}$ (see page 181).

Distributed lock $d$ protects access to service $s$. Users employ a locking protocol to acquire access to the service. In the unlocked state ($S_1$) $d$ can react to the following events:

- If lock $d$ receives a lock(p) request, $d$ assigns the lock to the caller (a lock coordinator). Lock $d$ creates and initializes an instance of service $s \in \text{Service}$, then sends a primary acknowledgement locked(s) to the lock coordinator; returning the identity of service $s$. The service is granted to the lock coordinator for a fixed period of time. Lock $d$ enables timer $t_{qt}$ to enforce the quantum then transitions to the locked state ($S_2$).

- If lock $d$ receives an unlock() request from the last holder of the lock, $d$ responds with secondary acknowledgement unlocked_r(). This situation arises when unlocked() acknowledgements are lost.

- If lock $d$ receives an unlock() request from a caller which was not the previous holder of the lock, $d$ responds with message reject().

In the locked state ($S_2$) $d$ can react to the following events:

- If lock $d$ receives a lock(p) request from the holder of the lock, $d$ sends secondary acknowledgement locked_r(s) to the holder. This situation can arise when the primary acknowledgement locked(s) is lost.

- If lock $d$ receives a lock(p) request from a caller which does not hold the lock, $d$ responds with message reject(), indicating that the lock is currently held by another lock coordinator.

- If lock $d$ receives an unlock() request from the holder of the lock, $d$ sends primary acknowledgement unlocked() to the caller, discards timer $t_{qt}$, stops service $s$, then transitions to state $S_1$.

\[3\] In Listing D.7 on page 234, state $S_{rv} = S_1 \cup S_2$. 
Figure 7.8: A distributed lock \( d \in \text{DLock} \).

Table 7.8: Application DLOCK, transition relations for \( d \in \text{DLock} \).

<table>
<thead>
<tr>
<th>( q_{\text{current}} )</th>
<th>( \Sigma_{\text{DLock}} )</th>
<th>Condition</th>
<th>( q_{\text{next}} )</th>
<th>( K^+ )</th>
<th>( J^+ )</th>
<th>( X^+ )</th>
<th>( R )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>init(n)</td>
<td></td>
<td>( S_1 )</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>becomes ( S_1 )</td>
</tr>
<tr>
<td>( S_1 )</td>
<td>unlock()</td>
<td>( k = * )</td>
<td>( S_1 )</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>commit</td>
</tr>
<tr>
<td>( S_1 )</td>
<td>unlock()</td>
<td>( k \neq * )</td>
<td>( S_1 )</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>commit</td>
</tr>
<tr>
<td>( S_1 )</td>
<td>lock(p)</td>
<td>( h = \text{none} )</td>
<td>( S_2 )</td>
<td>{ s }</td>
<td>{ +\text{qt} }</td>
<td>{ }</td>
<td>becomes ( S_2 )</td>
</tr>
<tr>
<td>( S_2 )</td>
<td>lock(p)</td>
<td>( h = * )</td>
<td>( S_2 )</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>commit</td>
</tr>
<tr>
<td>( S_2 )</td>
<td>unlock()</td>
<td>( k = * )</td>
<td>( S_2 )</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>commit</td>
</tr>
<tr>
<td>( S_2 )</td>
<td>unlock()</td>
<td>( h \neq * )</td>
<td>( S_2 )</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>commit</td>
</tr>
<tr>
<td>( S_2 )</td>
<td>unlock()</td>
<td>( h = * )</td>
<td>( S_2 )</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>commit</td>
</tr>
<tr>
<td>( S_2 )</td>
<td>noquantum()</td>
<td>( h \neq \text{none} )</td>
<td>( S_1 )</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>becomes ( S_1 )</td>
</tr>
<tr>
<td>*</td>
<td>stop()</td>
<td></td>
<td>( S_1 )</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>terminate</td>
</tr>
</tbody>
</table>

- If lock \( d \) receives an unlock() request from the previous holder of the lock, \( d \) sends secondary acknowledgement unlocked\(_r\)() to the caller. This situation arises when unlocked() acknowledgements are lost.
- If lock \( d \) receives an unlock() request from a caller which is neither the current nor the previous holder of the lock, \( d \) responds with message reject().

\(^4\)h is the holder of the lock; \( k \) is the last holder of the lock.
If lock $d$ receives timeout message noquantum() from timer $t_{qt}$, $d$ sends message unlocked() to the holder of lock, discards timer $t_{qt}$, stops service $s$, then transitions to state $S_1$. This condition can occur when service user $u$ has not released the lock, when proxy $p$ has not completed access to service $s$, or if the node on which $l$ resides has left the configuration.

**Class Locker—Distributed Lock Coordinator (Listing D.8).** A communicating finite state machine illustrating the behaviour of $l \in \text{Locker}$ is shown in Figure 7.9. The transition relation for this CFSM is summarized in Table 7.9.

The lock coordinator $l$ translates a lock request from user $u$ (lock($d$, $p$)) into a a lock request (lock($p$)) for $d \in \text{DLock}$. At this point the lock coordinator enables retransmission timer $t_{lock}$ and enters state **Locking**. In state **Locking** the lock coordinator reacts to the following situations:

- Lock $d$ replies with acknowledgement locked($s$) indicating that the lock coordinator has acquired lock $d$. The lock coordinator sends message locked($s$) to user $u$, disables retransmission timer $t_{lock}$, then transitions to state **Locked**.

- Lock $d$ replies with acknowledgment locked_r($s$) indicating that the lock coordinator already holds lock $d$. This can occur if the initial locked($s$) acknowledgement was lost. The lock coordinator sends message locked($s$) to user $u$, then transitions to state **Locked**.

- Lock $d$ replies with message reject() indicating that another lock coordinator holds the lock. The requesting lock coordinator takes no action, allowing $t_{lock}$ to timeout, and generate a new coordinator lock request (lock($p$)).

- Timer $t_{lock}$ generates timeout message noresponse(). If the number of lock attempts has not exceeded the coordinators threshold, the lock coordinator resends request lock($p$) to $d$; otherwise, the lock coordinator issues message apperror(1) to the user.
Figure 7.9: A lock coordinator $l \in \text{Locker}$.

Table 7.9: Application DLOCK, transition relation for a lock coordinator $l \in \text{Locker}$.
While the lock coordinator is in state **Locked**, proxy $p$ accesses service $s$, and the lock coordinator expects the following additional events:

- Message `unlock()` from user $u$ indicates that proxy $p$ has completed its access to service $s$ and the user is releasing the lock. In this case the lock coordinator enables the retransmission time $t_{lock}$ and transitions to state **Release**.

- Message `unlocked()` from the lock $d$ indicating that user $u$ has exceeded its lock quantum. In this situation user $u$ loses the lock, and the lock coordinator sends message `expired()` to user $u$ the transitions to state **Idle**. Notice that $d \in \text{DLock}$ only sends a single `unlocked()` message when a user exceeds its lock quantum. If message `unlocked()` is lost, the lock coordination will remain in state **Locked** until the user $u$ issues an `unlock()` request.

- Message `locked_r(s)` from lock $d$. This condition can occur when multiple `lock(p)` messages were generated by $l$ (due to the loss of primary acknowledgement `locked(d)`) to acquire lock $d$. The lock coordinator takes no action.

In state **Release** the lock coordinator can respond to the following events:

- Message `unlocked()` from lock $d$ indicating that service $s$ is now available. The lock coordinator sends message `unlocked()` to user $u$, disables the retransmission timer $t_{lock}$, then enters state **Idle**.

- Message `unlocked_r()` from lock $d$ indicating that $l$ was the last coordinator to hold the lock. This message is generated in response to duplicate `unlock()` requests made by $l$ and indicates that acknowledgement `unlocked()` was lost. The lock coordinator sends message `unlocked()` to user $u$, disables the retransmission timer $t_{lock}$, then enters state **Idle**.

- Message `reject()` from lock $d$ indicating that lock coordinator $l$ no longer holds the lock.

- Timer $t_{lock}$ generates timeout message `noresponse()`. If the number of unlock attempts has not exceeded the coordinators threshold, the lock coordinator resends the `unlock()` request; otherwise, it issues message `apperror(2)` to the user.

---

$n$ is the current number of attempts to acquire lock, $n_2$ is the maximum number of attempts.
7.3.2 An Analysis of Application DLOCK

To examine the characteristics of application DLOCK, we designed a simulation consisting of four service users $u_i$ and associated lock coordinators $l_i$ on each of two nodes (Nodes 2 and 3), and a single distributed lock $d$ and associated service $s$ on a third node (Node 1). During the simulation each service user $u$ competes for distributed lock $d$ until it is acquired exactly $n = 256$ times. For this simulation we employ an ABP receiver as service $s$ and an ABP sender as proxy $p$ (see Section 7.2). When the lock is granted to service user $u$, a stream of data is generated by proxy $p$ and consumed by service $s$.

Our analysis relies on the monitoring of several enumerations. A complete summary of all enumeration results is provided in Appendix D.2. In the remainder of this section we examine the results for User 1 on Node 2.

**Lock invariant.** The distributed lock $d$ is the most critical component in this simulation. The lock grants exclusive access to the service $s$. We employ an enumeration constraint

$$\phi_1 \triangleq 0 \leq \#!\text{locked} - \#!\text{unlocked} \leq 1$$

to monitor the binary lock invariant for distributed lock $d$. Rows 1 and 2 of Table 7.10 indicate that this condition was satisfied in each of the five reported trials. In addition, rows 3 and 4 indicates that the ABP receiver $s$ did not detect any out of sequence messages.

**Failure modes of the service user.** In this simulation we expect lock coordinator $l$ to report two types of failures to user $u$:

- message `expired()` indicates that the lock quantum expired before data transfer was complete, and

- message `apperror()` indicates that lock coordinator $l$ (on behalf of service user $u$) was unable to acquire or release lock $d$ within $n_1 \times n_2$ milliseconds (see Section 7.3.1.1).

We employ two enumerations

$$\phi_8 \triangleq \#\text{?expired}$$
$$\phi_9 \triangleq \#\text{?apperror}$$
to monitor these failures. Row 5 of Table 7.10 indicates that lock $d$ interrupted 3%–8% of the transfers. Row 6 of Table 7.10 indicates that lock coordinators fail to acquire or release the lock $d$ 3%–7% of the time.

Handling dropped control messages. The dropping of acknowledgements (locked(), locked_r(), unlocked(), and unlocked_r()) causes additional lock() and unlock() messages to be generated by a lock coordinator. Both situations are illustrated in Figure 7.10. In the present study we set the drop probabilities for messages locked(s), locked_r(s), unlocked(), and unlocked_r() to 0.01, to test...
whether a lock coordinator \( l \) correctly handles dropped control messages from lock \( d \).

We employ the following enumerations

\[
\begin{align*}
\phi_2 & \triangleq \#!\text{lock} \\
\phi_3 & \triangleq \#!\text{unlock} \\
\phi_4 & \triangleq \#?\text{locked} \\
\phi_5 & \triangleq \#?\text{locked}_r \\
\phi_6 & \triangleq \#?\text{unlocked} \\
\phi_7 & \triangleq \#?\text{unlocked}_r
\end{align*}
\]

to monitor the effect of dropped control messages on the lock coordinator.

Row 7 in Table 7.10 indicates that a significant number of \texttt{lock()} requests are required to acquire lock \( d \). On average, for the user described in this section, 13–15 \texttt{lock()} requests are made by the lock coordinator to acquire the lock. Row 6 indicates that 3%–7% of the time a lock could not be acquired after 40 \texttt{lock()} requests. Row 8 in Table 7.10 indicates that no more than one \texttt{unlock()} request is required to release a lock.

A lock coordinator generates spurious \texttt{lock()} and \texttt{unlock()} requests whenever the duration of its retransmission timer \( t_{\text{lock}} \) is set too short. In such an environment a greater number of secondary acknowledgements (\texttt{locked}_r() and \texttt{unlocked}_r()) are expected to be generated as duplicate \texttt{lock()} and \texttt{unlock()} requests are generated.

Rows 9–12 in Table 7.10 summarize the types and number of primary and secondary acknowledgements (control messages) received by a lock coordinator in response to \texttt{lock()} and \texttt{unlock()} requests. Rows 13–16 in Table 7.10 summarize the number of times these control messages are dropped during the simulation. We make the following observations concerning the frequency of control messages and the frequency with which they are dropped.

Rows 9 and 13 indicate that the total number of primary acknowledgements for request \texttt{lock()} is equal to \( n \), the number of complete \texttt{lock()}—\texttt{unlock()} cycles performed by locker \( l \) (Equation 7.7). In addition, rows 10 and 13 indicate that the number of secondary acknowledgements \texttt{locked}_r() is equal to the number of dropped \texttt{locked()} acknowledgements (Equation 7.8). These conditions hold for all 40 runs summarized in Table D.2.
These results indicate that the only source of lost `locked()` control frames in this simulation originate from `dprob` statements. When monitoring is disabled, the frequency of `locked_r()` control frames should be a good estimator of the rate of loss of primary acknowledgement `locked()`.

Rows 11 and 15 indicate that the total number of primary acknowledgements for request `unlock()` is equal to \( n \), the number of complete `lock()`—`unlock()` cycles performed by locker \( l \) (Equation 7.9). In addition, rows 12 and 15 indicate that the number of secondary acknowledgements `unlocked_r()` can be greater than or equal the number of dropped `unlocked()` acknowledgements (Equation 7.10). These conditions hold for all 40 runs summarized in Table D.2.

\[
\#?locked + \#\text{drop}(\text{locked}) = n \quad \text{(7.7)}
\]
\[
\#?\text{locked}_r = \#\text{drop}(\text{locked}) \quad \text{(7.8)}
\]

\[
\#?\text{unlocked} + \#\text{drop}(\text{unlocked}) = n \quad \text{(7.9)}
\]
\[
\#?\text{unlocked}_r \geq \#\text{drop}(\text{unlocked}) \quad \text{(7.10)}
\]

These results indicate that the only source of lost `unlocked()` control frames in this simulation originate from `dprob` statements. When monitoring is disabled, the frequency of `unlocked_r()` control frames will overestimate the rate of loss rate of primary acknowledgement `unlocked()`. In Equation 7.10 condition \( \#?\text{unlocked}_r \geq \#\text{drop}(\text{unlocked}) \) holds for 19 of the 40 runs summarized in Table D.2. This is a direct result of the bounded retransmission strategy used by the lock coordinator.
Table 7.10: Some characteristics of application DLOCK.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Class</th>
<th>Trial 1</th>
<th>Trial 2</th>
<th>Trial 3</th>
<th>Trial 4</th>
<th>Trial 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enumeration results for the DLOCK simulation (8 users, 8 lock coordinators)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1) Number of locking cycles</td>
<td>DLock</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>(2) Number of times lock invariant failed</td>
<td>DLock</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(3) Number of messages transmitted $p \in \text{ Proxy}$</td>
<td>Receiver</td>
<td>380018</td>
<td>381087</td>
<td>382458</td>
<td>381645</td>
<td>376879</td>
</tr>
<tr>
<td>(4) Number of out of sequence messages (Row 3)</td>
<td>Receiver</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Enumeration results for $u_1 \in \text{ User}$, Node 2, $n = 256$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(5) Number of quantum violations, $#？\text{ expired}$</td>
<td>User</td>
<td>19</td>
<td>9</td>
<td>13</td>
<td>15</td>
<td>21</td>
</tr>
<tr>
<td>(6) Failed to acquire lock, $#？\text{ apperror}$</td>
<td>User</td>
<td>13</td>
<td>18</td>
<td>14</td>
<td>17</td>
<td>12</td>
</tr>
<tr>
<td>Enumeration results for $l_1 \in \text{ Locker}$, Node 2, $n = 256$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(7) Number of lock requests, $#？\text{ lock}$</td>
<td>Locker</td>
<td>3478</td>
<td>3764</td>
<td>3614</td>
<td>3663</td>
<td>3297</td>
</tr>
<tr>
<td>(8) Number of unlock requests, $#？\text{ unlock}$</td>
<td>Locker</td>
<td>240</td>
<td>250</td>
<td>245</td>
<td>245</td>
<td>239</td>
</tr>
<tr>
<td>(9) Primary lock acknowledgements $#？\text{ locked}$</td>
<td>Locker</td>
<td>254</td>
<td>254</td>
<td>253</td>
<td>255</td>
<td>253</td>
</tr>
<tr>
<td>(10) Secondary lock acknowledgements, $#？\text{ locked}_r$</td>
<td>Locker</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>(11) Primary unlock acknowledgements, $#？\text{ unlocked}$</td>
<td>Locker</td>
<td>253</td>
<td>253</td>
<td>254</td>
<td>252</td>
<td>252</td>
</tr>
<tr>
<td>(12) Secondary unlock acknowledgements, $#？\text{ unlocked}_r$</td>
<td>Locker</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>(13) $#？\text{ drop(locked)}, p = 0.01$</td>
<td>Locker</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>(14) $#？\text{ drop(locked}_r), p = 0.01$</td>
<td>Locker</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(15) $#？\text{ drop(unlocked)}, p = 0.01$</td>
<td>Locker</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>(16) $#？\text{ drop(unlocked}_r), p = 0.01$</td>
<td>Locker</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
7.4 Summary

In this chapter we provide two examples of how applications can be modelled, specified, and analyzed using our approach for implementing distributed applications. Models for each application are first presented as a collection of message sequence charts and communicating finite state machines which capture the essential features of the protocols. The message sequence charts and communicating finite machines are converted to COOL class specifications, where message handlers implement the protocol and checkable properties are used to define the expected behaviour of selected actor classes. We collect the monitoring results (observed behaviour) from five trials for each application. Our analysis compares the observed behaviour with the expected behaviour for each application.

Application DABP. In Section 7.2 we describe application DABP, an implementation of the alternating bit protocol (ABP)

\[
DABP = \big| (s \mid t_{ack}) \mid u_s \big| \big| (r \mid u_r)
\]

where actors \( s \) and \( r \) provide an ABP service, actors \( u_s \) and \( u_r \) simulate service access points provided by the network layer, and timer \( t_{ack} \) is an acknowledgement timer associated with actor \( s \).

Our implementation of ABP enforces a sequence invariant

\[
\text{nf}(x[i]) = \text{sapb}(x[i]), \quad \forall i, \ 1 \leq i \leq n
\]

which reads “for a data stream of \( n \) frames the \( i \)-th frame produced by actor \( u_s \) is equal to the \( i \)-th data frame consumed by actor \( u_r \)”.

Using the following checkable properties

\[
\begin{align*}
\phi_1 & \triangleq g(seq) \ \text{assert}(\text{nfts} == \text{seq}) \\
\phi_2 & \triangleq \ ?nf \rightsquigarrow ?g : 300 \ \text{usec} \\
\phi_3 & \triangleq f(seq,x) \ \text{assert}(\text{fe} == \text{seq}) \\
\phi_4 & \triangleq 0 \leq \ ?f - \ ?f \leq 300 \ \text{usec}
\end{align*}
\]

we are able to; detect duplicate data and control frames (\( \phi_1 \) and \( \phi_3 \)), measure the throughput of data frames (\( \phi_4 \)), and estimate the end-to-end delay of control frames (\( \phi_2 \)). Our implementation of ABP
uses the \textit{dprob} clause and the \textit{abort} statement to inject a number of faults into the system. Both language features exercise the parts of the protocol that deal with recovery.

\textbf{Application DLOCK.} In Section 7.3 we describe application DLOCK, a simulation of a distributed lock protocol

\[
\text{DLOCK} = ((d \uparrow t_{qt}) \upharpoonright s) \upharpoonright (\Sigma_{i=1}^{n}((u_i \uparrow t_{req}) \upharpoonright (l_i \uparrow t_{lock}) \upharpoonright p_i))
\]

where \((d \uparrow t_{qt}) \upharpoonright s\) defines a lock service, and \((\Sigma_{i=1}^{n}((u_i \uparrow t_{req}) \upharpoonright (l_i \uparrow t_{lock}) \upharpoonright p_i))\) defines multiple service users. In the lock service, the distributed lock \(d\) controls access to service \(s\). The service user holding lock \(d\) has exclusive access to service \(s\) through proxy \(p\). The timer \(t_{qt}\) implements the lock quantum. In the user service, \(t_{lock}\) controls the retransmission characteristics of lock coordinator \(l\).

Our distributed lock simulation illustrates one mechanism for coordinating a group of actors, and demonstrates how \textit{bounded retransmission} can be used to make communication between peers reliable. Using the following checkable properties

\[
\begin{align*}
\phi_1 &\triangleq 0 \leq \#!\text{locked} - \#!\text{unlocked} \leq 1 \\
\phi_2 &\triangleq \#!\text{lock} \\
\phi_3 &\triangleq \#!\text{unlock} \\
\phi_4 &\triangleq \#?!\text{locked} \\
\phi_5 &\triangleq \#?!\text{locked}_r \\
\phi_6 &\triangleq \#?!\text{unlocked} \\
\phi_7 &\triangleq \#?!\text{unlocked}_r \\
\phi_8 &\triangleq \#?!\text{expired} \\
\phi_9 &\triangleq \#?!\text{apperror}
\end{align*}
\]

we are able to show; that the \textit{binary lock invariant} holds \((\phi_1)\), how to use enumerations to monitor the handling of primary acknowledgements \((\phi_2 - \phi_7)\), and how to use enumerations to collect information on the frequency of failures \((\phi_8 \text{ and } \phi_9)\).
Chapter 8

Conclusion

The modelling and implementation of distributed applications is often exacerbated by the absence of a common clock, lossy communications, the possibility of partial failures, and the requirement for a coordination strategy to guarantee the order of messages between communicating processes. These factors make it difficult to abstract distributed applications into a formal model. The primary motivation of this dissertation has been to provide a simple framework for implementing and experimenting with distributed applications. This framework deals with the modelling, specification, implementation and validation of distributed systems.

A formal model of distributed systems. Our formal model of distributed systems differs from [1] by accommodating a set of timers, and facilitating the atomic processing of input messages. We refer to this model as the Timed Actor model. Support for atomic message processing is provided by an intentions list which provides a record of the actions performed by an executing actor, and a set of commitment operations which implement a local checkpointing mechanism.

The coordination model [4] described here consists of two categories of coordinable objects, actors and timers; a coordination medium which supports point-to-point asynchronous communication between actors, or between an actor and a timer; and the coordination operators provided by an actor algebra called ACube. The timed actor semantics of ACube describes the semantics of commitment, asynchronous communication, actor creation, and timers.
A virtual machine to accommodate a timed actor language. We have described the characteristics of a virtual machine (VM) which implements the operational semantics of ACube and which accommodates a timed actor language. Our virtual machine provides an interface which maps the features of a timed actor language into equivalent virtual machine operations, e.g. new maps to vmnew, trigger maps to vmtrigger and send maps to vmsend. In our description of virtual machine operations we show how the semantics of virtual machine operations are supported by the timed actor semantics of ACube. We provide a pseudocode specification for the VM.

COOL—a timed actor language. COOL provides a high-level description language for specifying the interfaces, behaviour and coordination of actors; and provides features essential for an effective coordination language such as timer management, the ability to control the degree of concurrency, and the ability to distribute a computation.

The solution to some interesting language design issues are addressed by the informal semantics of COOL:

- All COOL language features map directly into VM instructions supporting our VM implementation.

- Since message transmission in COOL is typed and messages can only have one receiver, we provide an informal semantics for typechecking transmission statements to minimize the number of illegal messages reaching a receiver.

- We allow the definition of a message handler to include a clause to establish a drop probability for a message handler. During execution the run-time system probabilistically determines whether a dropable message will be received or abandoned. This is a simple mechanism for simulating message loss, allowing us to test how applications react to lost messages.

- We allow a class definition to specify a set of checkable properties to be monitored at run-time.

- We designed the commitment operators to support local error recovery. A practitioner who
designs COOL applications always has the choice of finalizing message processing, or abandoning message processing allowing an actor to return to its checkpointed state.

- In the design of COOL we make no provision for declaring, raising or handling run-time exceptions. Exceptions are handled transparently by our VM implementation.

**Property driven specification and analysis.** Formal approaches to the analysis of concurrent systems generally involve two notations: a notation for describing structural changes in a system, and a notation for expressing system properties. In our *Timed Actor model*, these notations are provided by *ACube*, an *asynchronous actor algebra*; and a grammar for expressing *checkable properties* in our *timed actor language*. Our approach is a property driven technique for specifying, monitoring and analyzing the performance and coordination properties of actors.

We allow the specification of an actor class **C** to describe the *expected behaviour* of an actor \( a \in C \). When a COOL compiler encounters a definition for a checkable property in a specification it instruments the implementation to enable monitoring of the specified checkable property. When the implementation is executed the VM and *online monitor* log event traces to an in-memory database. When execution is complete the VM prepares a log of the *observed behaviour* of all monitored actors. Logs created by the VM are analyzed by an *offline trace analyzer* ([see 13, Section B.2 on page 100]) which reports when the observed behaviour of each monitored actor differs from the expected behaviour defined in the actor class specification.

Our strategy of *online monitoring* coupled with *offline trace analysis* is selected to minimize the impact of monitoring on the performance of monitored actors. We measured the effect of monitoring on the running time of our DABP application. We found that the evaluation of a message preconditions requires about 2 \( \mu s \), collecting communication events for a follows constraint requires about 3 \( \mu s \), and collecting communication events for a timing constraint requires about 3 \( \mu s \). For the DABP application, where the mean response time of frames is reported to be approximately 170 \( \mu s \), the effect of monitoring is minimal.
**Case Studies.** We demonstrate the usefulness of our approach to distributed application development by conducting two case studies: an implementation of alternating bit protocol, and a simulation of a distributed locking protocol. Implementations of the protocols are provided using the cool2C translator described in [13, 14], and a C/C++ version of the VM described in [15].

The checkable properties used by our implementation of the alternating bit protocol were able to: verify the sequence invariant for ABP, detect duplicate data and control frames, measure the throughput of data frames, and estimate the end-to-end delay of control frames.

The checkable properties used by our implementation of a distributed locking protocol were able to: verify the binary lock invariant for the locking protocol, verify the handling of primary and secondary acknowledgements in the locking protocol, and collect information on the failure frequency of lock requests.

### 8.1 Contributions

Our contributions to this area of research include: extending the Actor model [1] by providing timers, and atomic message processing with local checkpointing; describing a timed actor semantics (ACube), which enables the specification of a timed actor language, and the specification of a virtual machine to accommodate our timed actor language; the specification of the syntax and informal semantics of COOL, a timed actor language whose implementation is described in [13, 14]; the incorporation of testing and monitoring features in a class specification; the development of a property driven technique for specifying, monitoring and analyzing the performance and coordination properties of actors; and the demonstration of the usefulness of our approach by applying it to a number of case studies.

### 8.2 Future Work

**Extending COOL.** Much of our future work for the Timed Actor model relates to extending the expressiveness of COOL, our timed actor language. We describe two new features which preserve our timed actor semantics: inheritance, which modifies the semantics of COOL; and message variables,
which modifies the semantics of our virtual machine and COOL.

Inheritance is a mechanism which makes it possible to define new actor classes by combining and specializing existing actor classes. In the following example class P inherits some of its behaviour from classes Q and R.

```plaintext
actor class P (Q x ; R y) { 
  msg a = x.g(int, int); 
  msg b = y.f(bool, actor); 
  msg c() { ... } 
  ...
}
actor class U { 
  P p; 
  msg h() { 
    p = new P; 
    a(1,2) => p; 
    ...
  }
}
```

In this example inheritance is used to combine the public interfaces of actor classes Q and R. The interface of a derived class (P) is defined by renaming message handlers in the inherited classes (msg a = x.g(int, int)), by redefining message handlers in the inherited classes, or by defining new message handlers.

ACube and COOL provide a necessary and sufficient semantics for managing actors and timers. In [5] Frølund introduces synchronizers as a mechanism for manipulating activation messages. For example, if a message arrives too early its activation can be delayed until a boolean expression associated with the message handler evaluates to true. Presently, in COOL, early messages can only be handled by storing the context of a communication in the local state of the activated actor.

A more useful approach for COOL would be to allow a programmer to have direct access to activation messages. Message variables are one facility for providing this type of access. In the following code segment the declaration ‘Q buf qb’ introduces message variable qb which can reference any activation message originating from an actor q ∈ Q.

```plaintext
actor class P { 
  Q buf qb; // A message variable
  ...
}
```
Q q = new Q;
...

msg f() from q buf qt { // Latch activation message
    ...
    qb = qt; // Store activation message
    commit;
}

msg h() {
    qb ⇒ q; // Deferred transmission
    qb = nil; // Release activation message
}

In our example message handler f() is provided with a guard which latches any message f which originates from actor q. If message handler f() determines that message f is early, the active actor acquires the activation message using the assignment qb = qt. In a latter activation of message handler h() our deferred message is transmitted to actor q (qb => q).

**Global checkable properties.** Presently in our *Timed Actor model* checkable properties are defined as part of individual actor class specifications; there is no support for specifying and monitoring communication events occurring in two or more local configurations. Supporting the specification and monitoring of checkable properties within a global configuration requires a modification of our property grammar (see Figure 6.2). For example, in application DABP we might like to measure the end-to-end delay of data frames between sender s and receiver r using the following timing constraint

$$\phi \triangleq 50 \leq {!f@s} - {!f@r} \leq 100 \text{ usec}$$

where {!f@s} identifies the transmission of message f() by sender s, and {!f@r} identifies the reception of message f() by receiver r. Our *Timed Actor model* is easily modified to instrument and monitor these types of communication events. The difficulty comes in providing our virtual machine with a *global clock* with the same resolution and accuracy as our present local master-clocks. In [21, 27] *globally synchronized clocks* are proposed as a global time reference for RTSynchronizers, but we have been unable to locate any cases studies evaluating their accuracy.
**Programming platforms.** The development platforms (primarily Linux and Windows® workstations) used to prototype our development tools (cool2C compiler, virtual machine for *timed actor semantics*, offline trace analyzer for *timed actor semantics*), and to develop our approach for implementing distributed applications, were chosen primarily on the basis of availability and cost-effectiveness. The next phase of our research involves migrating our virtual machine to an embedded platform. Migration to a new host platform requires a host operating system with basic timing, communication, and memory management services.
References


Appendix A

Actor Semantics

A.1 Illustrating Actor Interactions with Message Sequence Charts

In our Timed Actor model we use message sequence charts [7] to illustrate the interactions between actors in a global configuration. The message sequence charts illustrated in this dissertation use the typesetting tool described in [35]. A typical message sequence chart (MSC) generated by this tool is shown in Figure A.1.

Actor instances. Actor instances are represented by an instance head, an instance axis, and an instance end. The instance head is depicted as an open box and is labelled with the name of the actor instance. The instance end is depicted either by a filled box or a cross. When an actor terminates its instance end is represented by a cross.

Timer instances. Starting a timer instance is shown as a horizontal line connecting an hour glass symbol to the instance axis of the associated actor. The hour glass is labelled with the name of the timer instance. Discarding a timer is shown as a horizontal line connecting a cross to the instance axis of the associated actor. The cross is labelled with the name of the timer instance. A timeout is shown as a horizontal arrow connecting an hour glass to the instance axis of the associated actor.
Message passing. Messages are denoted by arrows originating from the instance axis of a sending actor to the instance axis of a receiving actor. Messages are labelled with their message name. A lost message is denoted by an arrow originating from the instance axis of a sending actor and ending in a small filled circle.

Messages originating from the left or right side of a MSC frame and ending at the instance axis of a receiving actor denote messages originating from actor instances external to the current frame.
(sometimes referred to as the *environment*). Messages ending at the left or right side of a MSC frame and originating from the instance axis of a sending actor denote messages to be delivered to actors instances *external* to the current frame.

### A.2 Actors as Communicating Finite State Machines

An actor instance is modelled as a *timed communicating finite state machine* (TCFSM). A TCFSM is a labelled directed graph with nodes representing conceptual states and edges labelled with the name of the message accepted and the names of the messages generated in those states [8, 9, 10]. Figure A.2 illustrates a typical TCFSM; where *timers* are depicted as small extrusions in the conceptual states where they are active. In Figure A.2 there are two distinguished states, the initial state **Start** and the final state **0**. When an edge is both accepting and generating the label takes the form $\sigma_i/\sigma_{o_1}(v_1), \sigma_{o_2}(v_2), \cdots$ where $\sigma_i$ is an input message and $\sigma_{o_1}(v_1), \sigma_{o_2}(v_2), \cdots$ is a finite set of output messages with parameters $v_1$ and $v_2$. We describe the communication behaviour of a TCFSM as *asynchronous*, because any number of messages can be generated during one transition.

**Example: A typical TCFSM.** Formally, let $\Sigma$ be the set of input message names accepted by an actor, then a TCFSM $G$ is a 6-tuple $(Q, Q_f, q_0, T, \Sigma, B)$ where $Q$ is a set of conceptual states, $Q_f \subseteq Q$ is a set of final states, $q_0$ is a distinguished initial state, $T$ is a set of timers, and $B$ describes the behaviour of an actor

$$B = K \times X \longmapsto K \times \mathcal{P}(K) \times \mathcal{P}(J) \times X^* \times R$$

(A.1)

where $K$ is the set of all possible actor instances, $\mathcal{P}(K)$ is the set of all finite subsets of actor instances, $\mathcal{P}(J)$ is the set of all finite subsets of timer instances, $X^*$ is all possible sequences of output messages, and $R$ is the set of all possible commitment operations. Equation A.1 is described on page 30.
All actor instances modelled by the TCFSM in Figure A.2 can be described by
\[ G = (Q, Q_f, q_0, T, \Sigma_G, B) \] where (assuming a destination actor \( a \) for all generated messages)

\[ Q = \{ \text{Start}, q_1, q_2, q_3, 0 \}, \quad Q_f = \{ 0 \}, \quad q_0 = \text{Start}, \quad T = \{ t \}, \]
\[ \Sigma_G = \{ m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8, m_9 \}, \text{ and} \]
\[ B \quad \text{is the transition relation summarized in Table A.1.} \]

---

**Table A.1: Transition relation for actor \( g \in G \).**

<table>
<thead>
<tr>
<th>( q_{\text{current}} )</th>
<th>( \Sigma_G )</th>
<th>( q_{\text{next}} )</th>
<th>( K^+ )</th>
<th>( J^-/J^+ )</th>
<th>( X^+ )</th>
<th>( R )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>( m_1() )</td>
<td>( q_1 )</td>
<td>{ }</td>
<td>{ }</td>
<td>[ ]</td>
<td>becomes ( q_1 )</td>
</tr>
<tr>
<td>Start</td>
<td>( m_5() )</td>
<td>( 0 )</td>
<td>{ }</td>
<td>{ }</td>
<td>[ ]</td>
<td>terminate</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( m_2() )</td>
<td>( q_2 )</td>
<td>{ }</td>
<td>{ +t }</td>
<td>[ ]</td>
<td>becomes ( q_2 )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( m_3() )</td>
<td>( q_3 )</td>
<td>{ }</td>
<td>{ }</td>
<td>[ ]</td>
<td>becomes ( q_3 )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( m_6() )</td>
<td>( 0 )</td>
<td>{ }</td>
<td>{ }</td>
<td>[ a \cdot n_1(x), a \cdot n_2(y) ]</td>
<td>terminate</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( m_4() )</td>
<td>( q_3 )</td>
<td>{ }</td>
<td>{ -t }</td>
<td>[ ]</td>
<td>becomes ( q_3 )</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( m_7() )</td>
<td>( 0 )</td>
<td>{ }</td>
<td>{ -t }</td>
<td>[ a \cdot n_1(x) ]</td>
<td>terminate</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( m_9() )</td>
<td>( q_1 )</td>
<td>{ }</td>
<td>{ }</td>
<td>[ ]</td>
<td>becomes ( q_1 )</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>( m_8() )</td>
<td>( 0 )</td>
<td>{ }</td>
<td>{ }</td>
<td>[ ]</td>
<td>terminate</td>
</tr>
</tbody>
</table>

In Table A.1 columns \( q_{\text{current}} \) and \( q_{\text{next}} \) show the current and next conceptual state of \( G \), column
\( \Sigma_G \) shows the legal input messages for \( G \), column \( K^+ \) shows the new actors created by an instance of \( G \), column \( J^+ / J^- \) indicates which timers are created (\( +t \)) or destroyed (\( -t \)) by an instance of \( G \), column \( X^+ \) shows the message sequences generated by an instance of \( G \), and \( R \) shows the replacement behaviour of \( G \).
Appendix B

Details of a Virtual Machine Implementation

B.1 Kernel Syntactic Domains

Names play an important role in the semantics of COOL specifications. Specifications may introduce message names, \( f, g, h \in \Sigma \); the names of conceptual states, \( q \in Q \); the names of actor classes, \( C \in C \); the names of local state variables, \( s \in S \); and, the names of message parameters, \( v \in V \).

In the kernel, object identities play a significant role in locating and discriminating objects. The kernel employs actor identities, \( a \in A \); timer identities, \( t \in T \); message identities, \( d \in M \); actor class identities, \( c \in C \); and node identities, \( n \in N \).

The syntactic domains used to describe our kernel are summarized Table B.1. Many of these domains are introduced in Chapters 2 and 3 to describe ACube timed actor semantics. A number of functions used to describe actions performed by the kernel are defined in Table B.2.

B.1.1 Object Identities

Actor and timer identities. Fresh actor and timer identities are generated by the kernel in response to the \texttt{vmnew} and \texttt{vmtrigger} operations. Function \texttt{fresh}(n,C) generates a fresh actor identity for an actor instance with class identity \( c = \text{convcid}(C) \) that will reside on node \( n \).
\[ a \in A \{ \text{instance number} \} \]

Function \( \text{fresh}(n, \text{Timer}) \) generates a fresh timer identity for a timer instance with class identity \( c = \text{convcid}(\text{Timer}) \) that will reside on node \( n \).

\[ t \in T \{ \text{instance number} \} \]

Function \( \text{fresh}(n, C) \) returns the value \textbf{none} if it is unable to generate a fresh actor or timer identity.

\textbf{Message identities.} Message identities encode class and message names. This allows the kernel to distinguish messages with the same name defined in two or more actor class definitions. The function \( \text{convmid}(\sigma, C) \) converts message name \( \sigma \), defined in actor class \( C \), to a message identity \( d \).

\[ d \in M \{ \text{sequence number} \} \]

\textbf{Function lookup()}. Actor and timer identities are used as keys to quickly locate objects in lists \( \mathcal{K} \), \( J \), and \( S \) (see page 75). The function \( \text{lookup}(i, l) \) returns a reference (domain \( Rf \)) to the element in list \( l \) which matches key \( i \). For example an actor identity \( a \) can be used to locate an actor instance in list \( \mathcal{K} \), \( k \leftarrow \text{lookup}(a, \mathcal{K}) \); and a timer identity \( t \) can be used to locate a timer instance in list \( J \), \( j \leftarrow \text{lookup}(t, J) \). Function \( \text{lookup}(i, l) \) returns the value \textbf{nil} if \( i \) cannot be found on list \( l \).
Table B.1: Syntactic domains used to describe the kernel.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object Identities</strong></td>
<td></td>
</tr>
<tr>
<td>$A$</td>
<td>An actor identity, $a \in A$.</td>
</tr>
<tr>
<td>$T$</td>
<td>A timer identity, $t \in T$.</td>
</tr>
<tr>
<td>$M$</td>
<td>A message identity, $d \in M$.</td>
</tr>
<tr>
<td>$N$</td>
<td>A node identity, $n \in N$.</td>
</tr>
<tr>
<td>$C$</td>
<td>A class identity, $c \in C$.</td>
</tr>
<tr>
<td><strong>Names</strong></td>
<td></td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>The name of a message, $f, g, h \in \Sigma$.</td>
</tr>
<tr>
<td>$Q$</td>
<td>The name of a conceptual state, $q \in Q$.</td>
</tr>
<tr>
<td>$\mathcal{C}$</td>
<td>The name of an actor class, $C \in \mathcal{C}$.</td>
</tr>
<tr>
<td><strong>Object Instances</strong></td>
<td></td>
</tr>
<tr>
<td>$K$</td>
<td>An actor instance, $k \in K$.</td>
</tr>
<tr>
<td>$J$</td>
<td>A timer instance, $j \in J$.</td>
</tr>
<tr>
<td>$X$</td>
<td>A message, $m \in X$.</td>
</tr>
<tr>
<td>$S$</td>
<td>The local state of an actor or a timer, $s \in S$.</td>
</tr>
<tr>
<td>$V$</td>
<td>The user-defined message parameters, $v \in V$.</td>
</tr>
<tr>
<td><strong>List and Reference Types</strong></td>
<td></td>
</tr>
<tr>
<td>$L$</td>
<td>An ordered or unordered list, $l \in L$.</td>
</tr>
<tr>
<td>$Rf$</td>
<td>A reference to a object instance, $o \in Rf$.</td>
</tr>
<tr>
<td><strong>Miscellaneous Types</strong></td>
<td></td>
</tr>
<tr>
<td>$P$</td>
<td>A timer type, $p \in {\text{oneshot, periodic}}$.</td>
</tr>
<tr>
<td>$R$</td>
<td>A commitment action, $r \in {\text{becomes, abort, terminate}}$.</td>
</tr>
<tr>
<td>$U$</td>
<td>A configuration update action, $u \in {\text{commit, rollback}}$.</td>
</tr>
<tr>
<td>$Y$</td>
<td>The type of an event generated by the active actor, $y \in Y$.</td>
</tr>
<tr>
<td>$E$</td>
<td>Exceptions, $e \in {\text{NOHANDLER, MGFAILED, NOSTATE, TGFAILED, NOTARGET, NORESOURCE}}$.</td>
</tr>
<tr>
<td>$N$</td>
<td>The natural numbers, $0, 1, 2, \ldots$.</td>
</tr>
<tr>
<td>$B$</td>
<td>The boolean values, $\text{true}$ and $\text{false}$.</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>Describes a function with no parameters or no return value.</td>
</tr>
</tbody>
</table>

### B.1.2 Object Instances

The kernel manipulates several categories of objects: actor instances, timer instances, messages, the local state of actors or timers, and user-defined message parameters.
**Actor instance.** In the kernel, an *actor instance* (see Section 2.1.1) takes the form

\[
k \in K \left\{ \begin{array}{cc|cc|c}
  a & q & s & s_2 & c \\
\end{array} \right. \]

where, \(a\) is an actor identity, \(q\) is a conceptual state, \(s\) is a reference to the private local state of \(a\), \(s_2\) is a reference to the checkpointed private local state of \(a\), and \(c\) is a reference to the *class table entry* for \(a\) (see Section 4.2). The `vmnew` operation (see Section 4.5.1) allocates actor instances, denoted \(k \leftarrow \text{allocate}(K)\), where \(k\) is an object reference of type \(K\&\).

**Actor local state.** In ACube semantics the local state of an actor, denoted \(s, s', s'', \cdots\), is viewed either as a scalar instance variable or as a *record* of instance variables. In the kernel \(s\) is manipulated like a Pascal record. We access members of the local state using the *dot* (.) *operator*. For example, we access member `self` of \(s\) as \(s\).self.

The local state of an actor consists of a set of system-defined instance variables maintained by the kernel, and a set of user-defined instance variables managed by an actor instance. The system-defined instance variables `owner`, `self`, `current`, and `now` have been defined in Section 5.1.4 on page 106. These instance variables are used so frequently in COOL it is customary to write them as keywords.
The vmnew operation allocates actor local state, denoted \( s \leftarrow \text{allocate}(S_X) \), where \( s \) is an object reference of type \( S\& \). A second local state object, denoted \( s_2 \leftarrow \text{allocate}(S_X) \) is allocated for local checkpointing.

**Timer instance.** In the kernel, a timer instance (see Section 2.2.1) takes the form

\[
j \in J \left\{ \begin{array}{cccccc}
  t & q & s & c & j' & \text{enabled}
\end{array} \right. \]

where, \( t \) is a timer identity, \( q \) is a conceptual state, \( s \) is a reference to the private local state of \( t \), \( c \) is a reference to the class table entry of class Timer (see Section 4.2), \( j' \) is a reference to a second timer instance required to support retriggering, and \( \text{enabled} \) is a boolean flag which determines if \( t \) is enabled. The vmtrigger operation (see Section 4.5.2) allocates timer instances, denoted \( j \leftarrow \text{allocate}(\beta) \), where \( j \) is an object reference of type \( J\& \).

**Timer local state.** As specialized actors, the local state of a timer contains a set of system-defined instance variables maintained by the kernel, and a set of instance variables required for managing a timer.

\[
s \in S_\beta \left\{ \begin{array}{ccc}
  \text{owner} & \text{self} \\
  \text{current} & \text{now} \\
  \text{message} \\
  \text{counter} & \text{type} & \text{period}
\end{array} \right. \]

\[
\left\{ \begin{array}{c}
  \text{System-Defined}
\end{array} \right. \left\{ \begin{array}{c}
  \text{Timer-Specific}
\end{array} \right. \]

The vmtrigger operation allocates timer local state, denoted \( s \leftarrow \text{allocate}(S_\beta) \), where \( s \) is an object reference of type \( S\& \). After initialization, instance variable \( \text{type} \) will hold one of oneshot or periodic, and instance variable \( \text{period} \) will hold the duration or period of the timer. Instance variable \( \text{counter} \) holds the current value of the timer and is reloaded as necessary from instance variable \( \text{period} \). The local state of every timer instance contains a reference to a message buffer \( \text{message} \). \( \text{message} \) holds a formatted message to be sent to the timers owner when the timer expires.
**Messages and message parameters.** In ACube semantics a message \( m \) is represented by a 4-tuple \( (\star, a, \sigma, v) \) where \( \star, a \in A \) are the source and destination actor identities respectively, \( \sigma \) is the input message name, and \( v \in V \) is the message contents. The message contents (message parameters), denoted \( v, v', v'', \ldots \), is viewed either as a scalar instance variable or as a record of instance variables. In the kernel it is convenient to view message parameters as a record of instance variables. We access message parameters using the *dot (.) operator*. For example, we access member \( \text{mid} \) of \( m \) as \( m.\text{mid} \).

Operations \( \text{vmtrigger}, \text{vmsend}, \) and \( \text{vmforward} \) allocate messages, denoted \( m \leftarrow \text{allocate}(B_V) \), where \( m \) is an object reference of type \( X& \).

**Selector functions.** A number of selector functions are available to access the current state of actor \( a \); function \( \text{state}(a) \) returns a reference to the private local state of actor \( a \), function \( \text{current}(a) \) determines the current conceptual state of actor \( a \), function \( \text{node}(a) \) returns the identity of the node \( (n) \) on which actor \( a \) resides, and function \( \text{class}(a) \) returns the class identity \( (c) \) of actor \( a \).

A number of selector functions are available to access the current state of timer \( t \); function \( \text{expired}(t) \) determines if timer \( t \) has timed out, function \( \text{type}(t) \) returns type of timer \( t \), and function \( \text{owner}(t) \) can be used to determine the owner of timer \( t \).

A number of selector functions are available to access the message referenced through \( m \); function \( \text{source}(m) \) returns the identity of the sender of message \( m \), \( \text{dest}(m) \) returns the identity of the des-
tination of message \( m \), \( \text{mid}(m) \) returns the identity of message \( m \), \( \text{clock}(m) \) extracts the timestamp from message \( m \), and \( \text{params}(m) \) returns a reference to the message contents of \( m \).

### B.1.3 Object References

A *reference* is an abstraction of a hardware address. Let \( Rf \), with typical elements \( o, o', o'', \cdots \) denote object references. The kernel utilizes object references to access domains \( K, J, X, S, \) and \( V \). We denote a domain of references by appending the & operator to the domain symbol. For example, \( K\& \) is the domain of references to actor instances, \( J\& \) is the domain of references to timer instances, and \( X\& \) is the domain of references to messages.

### B.1.4 Object Lists

The VM organizes kernel objects into two types of lists \( (L) \). The first type of list, which includes \( K, J, S_K, S_J, B_V \) and \( B_N \); is unordered and contains a fixed number of objects. \( K \) contains free and allocated actor instances; \( J \) contains free and allocated timer instances; \( S_K \) contains free and allocated local state objects for actors; \( S_J \) contains free and allocated local state objects for timers; and \( B_V \) and \( B_N \) contain free and allocated message buffers.

The second type of list, which includes \( X_L, X_I, X_O, I, R, \) and \( T \); is ordered and contains a variable number of *references to objects* in \( K, J, B_V \) and \( B_N \). \( X_L, X_I, X_O \) contain references to messages that will be processed by Dispatcher; \( R \) and \( T \) contain references to messages that will be processed by Dispatcher and the node; and \( I \) contains references to actor instances, timer instances and messages created by the active actor.

**List functions.** A number of functions are required to manage these lists. Function \( \text{allocate}(l) \) returns a reference to a free element on list \( l \in \{K, J, S_K, S_J, B_V, B_N \} \). Function \( \text{allocate}(l) \) returns \text{nil} if it is unable to allocate the necessary reference. Function \( \text{deallocate}(r, l) \) frees the list element referenced through \( r \) in list \( l \). The ordered lists \( X_L, X_I, X_O, I, R, \) and \( T \) are managed with a FIFO discipline. \( \text{head}(l) \) returns the first element of list \( l \), and \( \text{tail}(l) \) returns the tail of list \( l \). An ordered list \( l \) is represented as \( \text{head}(l) \bowtie\text{tail}(l) \), where the operator \( \bowtie \) specifies concatenation of lists. An
element $r$ is added to an ordered list $l$ by concatenation, $l \leftarrow r$.

Table B.2: Functions used in the description of the kernel.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functions to access a lists</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>allocate($l$)</td>
<td>Return a reference to a free element on list $l$.</td>
<td>allocate : $L \mapsto Rf$</td>
</tr>
<tr>
<td>deallocate($r$, $l$)</td>
<td>Free element referenced through $r$ on list $l$.</td>
<td>deallocate : $Rf \times L \mapsto \top$</td>
</tr>
<tr>
<td>head($l$)</td>
<td>Return the first element in list $l$.</td>
<td>head : $L \mapsto Rf$</td>
</tr>
<tr>
<td>tail($l$)</td>
<td>Return the tail of list $l$.</td>
<td>tail : $L \mapsto L$</td>
</tr>
<tr>
<td>lookup($i$, $l$)</td>
<td>Return a reference to the element in list $l$ with key $i$.</td>
<td>lookup : key $\times L \mapsto Rf$</td>
</tr>
<tr>
<td><strong>Conversion functions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fresh($n$, $C$)</td>
<td>Return a fresh actor identity for node $n$ and class $C$.</td>
<td>fresh : $N \times C \mapsto A$</td>
</tr>
<tr>
<td>convcid($C$)</td>
<td>Convert class name $C$ to class identity.</td>
<td>convcid : $C \mapsto C$</td>
</tr>
<tr>
<td>convmid($\sigma$, $C$)</td>
<td>Convert message name $\sigma$ to a message identity.</td>
<td>convmid : $\Sigma \times C \mapsto M$</td>
</tr>
<tr>
<td><strong>Functions to access actor instances (a)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>actors($e$)</td>
<td>Return a list of actor identifiers referenced in $e$.</td>
<td>actors : $Exp \mapsto L$</td>
</tr>
<tr>
<td>state($a$)</td>
<td>Return a reference to local state of actor $a$.</td>
<td>state : $A \mapsto Rf$</td>
</tr>
<tr>
<td>current($a$)</td>
<td>Return the current conceptual state of actor $a$.</td>
<td>current : $A \mapsto Q$</td>
</tr>
<tr>
<td>node($a$)</td>
<td>Return the identity of the node of actor $a$.</td>
<td>node : $A \mapsto N$</td>
</tr>
<tr>
<td>class($a$)</td>
<td>Return the class identity of actor $a$.</td>
<td>class : $A \mapsto C$</td>
</tr>
<tr>
<td><strong>Functions to access timer instances (t)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>timers($e$)</td>
<td>Return a list of timer identifiers referenced in $e$.</td>
<td>timers : $Exp \mapsto L$</td>
</tr>
<tr>
<td>expired($t$)</td>
<td>Determine if timer $t$ has timed out.</td>
<td>expired : $T \mapsto \top$</td>
</tr>
<tr>
<td>type($t$)</td>
<td>Determine the type of timer $t$.</td>
<td>type : $T \mapsto P$</td>
</tr>
<tr>
<td>owner($t$)</td>
<td>Return the identity of the owner of timer $t$.</td>
<td>owner : $T \mapsto A$</td>
</tr>
<tr>
<td><strong>Functions to access messages (m)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>source($m$)</td>
<td>Return actor identity of the sender of message $m$.</td>
<td>source : $X \mapsto A$</td>
</tr>
<tr>
<td>dest($m$)</td>
<td>Return actor identity of the receiver of message $m$.</td>
<td>dest : $X \mapsto A$</td>
</tr>
<tr>
<td>mid($m$)</td>
<td>Return the identity of message $m$.</td>
<td>mid : $X \mapsto M$</td>
</tr>
<tr>
<td>clock($m$)</td>
<td>Retrieve logical clock from $m$.</td>
<td>clock : $X \mapsto N$</td>
</tr>
<tr>
<td>params($m$)</td>
<td>Return a reference to message contents of $m$.</td>
<td>params : $X \mapsto V$</td>
</tr>
<tr>
<td><strong>Logging Facility</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>StartLog()</td>
<td>Start the system logging facility.</td>
<td>StartLog : $\top \mapsto \top$</td>
</tr>
<tr>
<td>EndLog()</td>
<td>Stop the system logging facility.</td>
<td>EndLog : $\top \mapsto \top$</td>
</tr>
<tr>
<td>Log($y$, ...)</td>
<td>Write a record to the system log.</td>
<td>Log : $Y \times \cdots \mapsto \top$</td>
</tr>
<tr>
<td>cexit()</td>
<td>Check if application has terminated.</td>
<td>cexit : $\top \mapsto B$</td>
</tr>
</tbody>
</table>
B.2 A Virtual Machine Supporting Timed Actor Semantics

B.2.1 A Pseudocode for Describing a Virtual Machine

The implementations provided in Section B.2.2 are presented using a Pascal-like pseudocode summarized in Table B.3 [54]. This pseudocode provides a standard set of control flow statements and four types of routines: an algorithm to describe an algorithm, an operation to describe a method in the abstract kernel interface, a process which provides an implementation of an actor class, and a function to describe a function or procedure.

Table B.3: Pseudocode features for describing a kernel.

<table>
<thead>
<tr>
<th>Pseudocode Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Control Flow</strong></td>
<td></td>
</tr>
<tr>
<td>if &lt;condition&gt; then &lt;stmt&gt; else then &lt;stmt&gt;</td>
<td>Selection statement.</td>
</tr>
<tr>
<td>for each &lt;x&gt; do &lt;stmt&gt;</td>
<td>Repetition over a list or set.</td>
</tr>
<tr>
<td>while &lt;condition&gt; do &lt;stmt&gt;</td>
<td>Iterate while condition is true.</td>
</tr>
<tr>
<td>repeat &lt;stmt&gt; until &lt;condition&gt;</td>
<td>Repeat until condition is true.</td>
</tr>
<tr>
<td><strong>Routines</strong></td>
<td></td>
</tr>
<tr>
<td>algorithm &lt;name(p1,p2,…)&gt; &lt;stmt&gt; return(x1,x2,…)</td>
<td>An algorithm.</td>
</tr>
<tr>
<td>operation &lt;name(p1,p2,…)&gt; &lt;stmt&gt; return(x1,x2,…)</td>
<td>A VM operation.</td>
</tr>
<tr>
<td>process &lt;name(p1,p2,…)&gt; &lt;stmt&gt; return(x1,x2,…)</td>
<td>The implementation of an actor class.</td>
</tr>
<tr>
<td>function &lt;name(p1,p2,…)&gt; &lt;stmt&gt; return(x1,x2,…)</td>
<td>A general function.</td>
</tr>
<tr>
<td><strong>Data Manipulation</strong></td>
<td></td>
</tr>
<tr>
<td>with &lt;record&gt; do &lt;stmt&gt;</td>
<td>Record access x,y becomes y.</td>
</tr>
<tr>
<td>x ← e</td>
<td>Simple assignment statement.</td>
</tr>
<tr>
<td>s.x ← e</td>
<td>Assign expression e to member x of local state.</td>
</tr>
<tr>
<td>v.x ← e</td>
<td>Assign expression e to member x of message contents.</td>
</tr>
</tbody>
</table>
### B.2.2 Listings for a Virtual Machine

Table B.4: Pseudocode specification for a virtual machine supporting timed actor semantics.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Listings</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dispatcher-actor Interface</strong></td>
<td></td>
</tr>
<tr>
<td>Process DspProcess</td>
<td>Listing B.1 starting on page 217.</td>
</tr>
<tr>
<td>Algorithm CheckTimers</td>
<td>Listing B.1 starting on page 217.</td>
</tr>
<tr>
<td>Algorithm CheckInbound</td>
<td>Listing B.1 starting on page 217.</td>
</tr>
<tr>
<td>Algorithm CheckOutbound</td>
<td>Listing B.2 starting on page 218.</td>
</tr>
<tr>
<td>Algorithm CheckLocal</td>
<td>Listing B.2 starting on page 218.</td>
</tr>
<tr>
<td>Operation vmdispatch</td>
<td>Listing B.2 starting on page 218.</td>
</tr>
<tr>
<td>Operation vmprocess</td>
<td>Listing B.3 starting on page 219.</td>
</tr>
<tr>
<td><strong>Actor-kernel Interface</strong></td>
<td></td>
</tr>
<tr>
<td>Operation vmnew</td>
<td>Listing B.4 starting on page 220.</td>
</tr>
<tr>
<td>Operation vmnew'</td>
<td>Listing B.4 starting on page 220.</td>
</tr>
<tr>
<td>Operation vmtrigger</td>
<td>Listing B.4 starting on page 220.</td>
</tr>
<tr>
<td>Operation vmtrigger'</td>
<td>Listing B.5 starting on page 221.</td>
</tr>
<tr>
<td>Operation vmdiscard</td>
<td>Listing B.5 starting on page 221.</td>
</tr>
<tr>
<td>Operation vmdiscard'</td>
<td>Listing B.5 starting on page 221.</td>
</tr>
<tr>
<td>Operation vmsend</td>
<td>Listing B.5 starting on page 221.</td>
</tr>
<tr>
<td>Operation vmsend'</td>
<td>Listing B.6 starting on page 222.</td>
</tr>
<tr>
<td>Operation vmforward</td>
<td>Listing B.5 starting on page 221.</td>
</tr>
<tr>
<td>Operation vmbecomes</td>
<td>Listing B.6 starting on page 222.</td>
</tr>
<tr>
<td>Operation vmabort</td>
<td>Listing B.6 starting on page 222.</td>
</tr>
<tr>
<td>Operation vmterminate</td>
<td>Listing B.6 starting on page 222.</td>
</tr>
</tbody>
</table>
Listing B.1: DspProcess, CheckTimers, and CheckInbound—managing timers and inbound messages.

process DspProcess(s, m)

StartLog()
repeat
CheckTimers(J)
if CheckInbound(X_I, s) = true
then
    if CheckOutBound(X_O) = true
    then
        CheckLocal(X_L)
until cexit() = true
EndLog()
return (terminate, 0)

function DspConstructor(s)
// Initialize user-defined local state
s_u ← (cnt ← 0, freq ← 2)
return

algorithm CheckTimers(l)
for each j ∈ l
    t ← id(j)
a ← owner(t)
if expired(t)
do
    if a ∈ actors(K)
        m ← message(t)
vmdispatch(a, m, false)
    // Restart timer if necessary
    if type(t) = oneshot
        then
            L ← inc(L)
        vmdiscard(t, commit)
    else if type(t) = periodic
        then
            L ← inc(L)
        vmttrigger(nil, t, commit)
    else if a /∈ actors(K)
        then
            vmnotify(a, NOTARGET, 0)
vmdiscard(t, commit)
    then
        l ← tail(l)
a ← dest(m)
if a ∈ actors(K)
    then
        vmdispatch(a, m, true)
    then
        return (false)
else if a /∈ actors(K)
    then
        vmnotify(a, NOTARGET, 0)
deadallocate(m, BV)
    then
        return (true)

function Schedule(s)
with s
    cnt ← inc(cnt)
if (cnt mod freq = 0)
do
    then return (true)
else
    then return (false)

function DrainReceiveBuffer()
for each m_i ∈ R
    do
        L ← time of reception
        then
            L ← inc(max(L, clock(m_i)))
        X_I ⌢ mo
        deallocate(m_i, BV)
return
Listing B.2: CheckOutbound and CheckLocal—managing inbound and local messages; and vmdispatch—dispatch a ready actor.

function DrainTransmitBuffer()
    for each \( m_i \in X_O \)
        \( L \leftarrow \text{inc}(L) \)
        \( m_o \leftarrow \text{allocate}(B_N) \)
        \( \text{copy}(m_i, m_o) \)
        do // Insert L into outgoing message
            \( m_o, \text{clock} \leftarrow L \)
            \( T \leftarrow m_o \)
            \( \text{deallocate}(m_i, B_V) \)
    return

algorithm CheckOutbound(l)
    if head(l)
        then \( \text{DrainTransmitBuffer()} \)
        return \( \text{false} \)
    then \( \text{if } X_O \text{ is empty} \)
        \( \text{return } \text{true} \)
    algorithm CheckLocal(l)
    \( m \leftarrow \text{head}(l) \)
    if \( m = \text{nil} \)
        then return
    \( l \leftarrow \text{tail}(l) \)
    \( a \leftarrow \text{dest}(m) \)
    if \( a \in \text{actors}(X) \)
        then vmdispatch(\( a, m, \text{true} \))
    else if \( a \notin \text{actors}(X) \)
        then vmnotify(\( a, \text{NOTARGET}, 0 \))
        then \( \text{deallocate}(m, B_V) \)
    return

operation vmdispatch(\( a, m, b \))
    // \( a = \text{current actor}, m = \text{current message}, b = \text{true}, \text{deallocate } m \)
    \( k \leftarrow \text{lookup}(a, X) \)
    if \( k = \text{nil} \)
        then \( \text{deallocate}(m, B_V) \)
        \( \text{return} \)
    vncachepoint(\( a \))
    vmactive(\( a, k, s, m \))
    // Return message buffer to buffer pool
    if \( b = \text{true} \)
        then \( \text{deallocate}(m, B_V) \)
    \( \text{return} \)

operation vmactive(\( a, s, m \))
    // \( a = \text{current actor}, s = \text{local state}, m = \text{current message} \)
    \( k \leftarrow \text{lookup}(a, X) \)
    \( j \leftarrow \text{nil} \)
    \( k.s.now \leftarrow T \)
    // Activate actor
    \( L \leftarrow \text{inc}(L) \)
    \( r.q \leftarrow k.c.\text{process}(s, m) \)
    // Reached commitment point
    case \( r \)
        of \( \text{: becomes :} \)
            \( \text{vmprocess}(J, \text{commit}) \)
            \( k.q \leftarrow q \)
        of \( \text{: terminate :} \)
            \( \text{vmprocess}(J, \text{commit}) \)
            \( \text{deallocate}(k.s, S_K) \cdot \text{deallocate}(k.s_2, S_K) \cdot \text{deallocate}(k,X) \)
        of \( \text{: abort :} \)
            \( \text{vmprocess}(J, \text{rollback}) \)
            \( \text{swap}(k.s, k.s_2) \)
    return

// Restore local state from checkpoint
// \text{swap}(k.s, k.s_2)
Listing B.3: \texttt{vmprocess}—an operation for processing an intentions list.

\begin{verbatim}
operation vmcheckpoint(a)
  // a=current actor
  \{ k ← lookup(a, \mathcal{K})
           \} copy(k.s,k.s2)
  \{ return \}

operation vmprocess(l, u)
  // l=an intentions list(l), u=configuration update action
  while x ← head(l)
  \{ case type(x) \}
  \{ of \}
  \begin{align*}
    &: K &:& \text{vmnew}'(b, u) \\
    &: J_t &:& \text{vmtrigger}'(j, j', t, u) \\
    &: J_d &:& \text{vmdiscard}'(t, u) \\
    &: X &:& \text{vmsend}'(\text{source}(m), m, u)
  \end{align*}
  \{ l ← tail(l) \}
  return
\end{verbatim}
Listing B.4: \texttt{vmnew}—an operation for actor creation; and \texttt{vmtrigger}—an operation for timer creation.

\begin{verbatim}
operation \texttt{vmnew}(a, C)  // Input: \texttt{a}=owner, \texttt{C}=class name  // Output: \texttt{b}=fresh actor identity belonging to class \texttt{C}  
L ← inc(L)  
b ← fresh(node(a), C)  
if \texttt{b} = \texttt{none}  
    \{ \vmnotify(a, \text{NORESOURCE}, \text{current}(a), 1) \}  
return (b)  
k ← allocate(\text{X})  
s, s’ ← allocate(S_X)  
// Initialize actor instance  
k ← (b, q ← \text{Start}, s, s’ \text{ClassTable[convcid(C)]})  
// Initialize system-defined local state  
s ← (\texttt{owner} ← a, \texttt{self} ← b, \texttt{current} ← k.q, \texttt{now}, u)  
j ← k  
return (b)  

operation \texttt{vmnew’}(b, u)  
k ← lookup(b, \text{X})  
\{  
\textbf{case} \texttt{u}  
\{  
\textbf{commit} :  
// Initialize \texttt{s.u}, user-defined local state  
// based on actor class definition  
k.c.constructor(k.s)  
\}  
\textbf{rollback} :  
\{  
deallocate(k.s, S_X) \cdot \text{dallocate}(k.s2, S_X) \cdot \text{dallocate}(k, \text{X})  
\}  
return \}
\end{verbatim}

operation \texttt{vmtrigger}(a, t, d, i, p)  // Input: \texttt{a}=owner identity, \texttt{t}=timer identity,  // \texttt{d}=message identity of timeout message,  // \texttt{i}=duration or period, \texttt{p}=type of timer \texttt{oneshot} or \texttt{periodic}  // Output: \texttt{t}=timer identity, \texttt{m}=reference to timeout message  
L ← inc(L)  
if \texttt{t} ≠ \texttt{none}  
\{ \texttt{j'} ← lookup(t, J) \}  
else \{ \texttt{j'} ← \text{nil} \}  
t ← fresh(node(a), \text{Timer})  
if \texttt{t} = \texttt{none}  
\{ \vmnotify(a, \text{NORESOURCE}, \text{current}(a), 2) \}  
\textbf{if} \texttt{j'} ≠ \texttt{nil}  
\{  
deallocate(f, m, \text{B_V}) \cdot \text{deallocate}(f' \cdot S)  
\textbf{then} \{  
\text{deallocate}(f', \text{J})  
\}  
\textbf{return} (t)  
\}  
\textbf{return} (t, m)
\end{verbatim}
Listing B.5: vmdiscard—an operation for timer destruction; and vmsend,vmforward—operations for message transmission.

```c
operation vmtrigger'((j',t,u))
{
    j ← lookup(t, J)
    case u
    of
        commit:
            if j' ≠ nil
                then {deallocate(j', BV) · deallocate(j, Sj)}
            of
                then j ← lookup(t, J)
                then {j.s.u.counter ← j.s.u.period
                        j.enabled ← true}
                of
                    deallocate(j, BV) · deallocate(j, Sj) · deallocate(j, J)
    // Restore t from j'.
    return
}

operation vmdiscard(a,t)
// Input: a=owner of timer, t=timer identity
    L ← inc(L)
    j ← lookup(t, J)
    j' ← j
    return

operation vmdiscard'(t,u)
if t ≠ none
    then return
    case u
    of
        commit:
            j ← lookup(t, J)
            if j ≠ nil
                then {deallocate(j, BV) · deallocate(j, Sj)}
        of
            then j ← lookup(t, J)
            then {j.s.u.counter ← j.s.u.period
                        j.enabled ← true}
            of
                deallocate(j, BV) · deallocate(j, Sj) · deallocate(j, J)
                // Restore t from j'.
                of
                    return

operation vmsend(a,b,d)
// Input: a=source actor, b=destination actor,
// d=identity of message being generated.
// Output: m=a reference to the output message buffer
    L ← inc(L)
    m ← allocate(BV)
    if m = nil
        then {vminotify(a, NORESOURCE, current(a), 3)}
    of
        return (m)
    // Initialize message buffer
    m ← (source ← a, dest ← b, mid ← d)
    j ← m
    return (m)

operation vmforward(m,b,d)
// Input: m=reference to the input message, b=destination actor,
// d=identity of message being generated.
// Output: none
    L ← inc(L)
    m ← allocate(BV)
    if m = nil
        then {vminotify(a, NORESOURCE, current(a), 3)}
    of
        return (m)
    copy(m, m)
    // Initialize message buffer
    m ← (dest ← b, mid ← d)
    j ← m
    return
```

221
Listing B.6: \texttt{vmbecomes}, \texttt{vmabort}, and \texttt{vterminate}—the kernel commitment operations.

\begin{verbatim}
operation \texttt{vmsend}(a, m, u)
\begin{cases}
  \text{commit} : \\
  \text{if node}(a) = \text{node}(\text{dest}(m)) \\
  \text{then } \{ x_L \leftarrow m \} \\
  \text{else } \{ x_O \leftarrow m \}
\end{cases}
\end{verbatim}

\begin{verbatim}
\texttt{vmbecomes}(q)
\begin{cases}
  L \leftarrow \text{inc}(L) \\
  \text{// Return to vmdispatch} \\
  \text{return } (\text{becomes}, q)
\end{cases}
\end{verbatim}

\begin{verbatim}
\texttt{vmabort}()
\begin{cases}
  L \leftarrow \text{inc}(L) \\
  \text{// Return to vmdispatch} \\
  \text{return } (\text{abort}, 0)
\end{cases}
\end{verbatim}

\begin{verbatim}
\texttt{vterminate}()
\begin{cases}
  L \leftarrow \text{inc}(L) \\
  \text{// Return to vmdispatch} \\
  \text{return } (\text{terminate}, 0)
\end{cases}
\end{verbatim}
Appendix C

COOL Event Logs

C.1 The Structure of COOL Event Log Records

The VM manages online monitoring. During execution all monitored events are stored as an in-memory database at each node participating in the application. When an application terminates the VM prepares a COOL event log.

This section describes the structure of the records created by the VM logging function Log(). Function Log() is used in every segment of monitoring code produced by the COOL compiler (see Section 6.3.3)

\[
\text{Log : } Y \times \cdots \mapsto V
\]

where, \( y \in Y \) identifies the type of an event log record, and \( \cdots \) is replaced with the context for a record of type \( y \).

C.1.1 Application Initialization Record

An application initialization record is logged when an application is activated. The record includes the initial value of the system logical clock \( L \), the current value of the system master-clock \( T \) (boot-time), and the identity of the node \( n \).

| ‘x’ | \( \text{L = 0} \) | \( T \) | \( n \) |
C.1.2 Timeout Record

A timeout record is logged each time an active timer times out. The record includes the current value of the system logical clock \( L \), the identity of the owner \( a \) of timer \( t \), the identity of the active timer \( t \), the identity of the timeout message handler \( m \), the duration (interval) of the timer \( n \), and the timer type \( p \) (oneshot or periodic).

<table>
<thead>
<tr>
<th>( 'j' )</th>
<th>( L )</th>
<th>( a )</th>
<th>( t )</th>
<th>( !m )</th>
<th>( n )</th>
<th>( p )</th>
</tr>
</thead>
</table>

C.1.3 Drop Record

A drop record is logged each time a received message satisfies the \( \text{dprob} \) clause of a message handler definition. The record includes the current value of the system logical clock \( L \), the identity \( a \) of the actor receiving the message, and the identity of the message \( m \) being dropped.

<table>
<thead>
<tr>
<th>( 'T' )</th>
<th>( L )</th>
<th>( a )</th>
<th>( m )</th>
</tr>
</thead>
</table>

C.1.4 Receive Record

A receive record is logged each time a monitorable message handler is activated. The record includes the current value of the system logical clock \( L \), the identity of the activated actor \( a \), the identity of the message handler \( m \), the identity of the actor \( b \) sourcing message \( m \), the value of the system master-clock \( T \) at activation (\text{now}), and the current conceptual state of the activated actor \( q \).

<table>
<thead>
<tr>
<th>( 'r' )</th>
<th>( L )</th>
<th>( a )</th>
<th>( ?m )</th>
<th>( b )</th>
<th>\text{now}</th>
<th>( q )</th>
</tr>
</thead>
</table>

C.1.5 Assertion Record

An assertion record contains the results of evaluating an assertion on entry to the message handler. Assertion records are generated only when an assertion fails. The record includes the identity of a message precondition \( \phi \), and the boolean constant \text{false}.

| \( 'A' \) | \( L \) | \( a \) | \( ?m \) | \( b \) | \text{false} | \( q \) |
C.1.6 Creation Record

A creation record is logged each time an actor instance is created using the `new` operation. The record includes the current value of the system logical clock $\mathbb{L}$, the identity of the owner $a$ of actor $b$, and the identity of the new actor instance $b$.

\[
\begin{array}{|c|c|c|}
\hline
'p' & \phi_i & \text{false} \\
\hline
\end{array}
\]

C.1.7 Trigger Record

A trigger record is logged each time a timer is created. The record includes the current value of the system logical clock $\mathbb{L}$, the identity of the owner $a$ of timer $t$, the identity of the fresh timer $t$, the identity of the timeout message handler $m$, the duration (interval) of the timer $n$, and the timer type $p$ (oneshot or periodic).

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
't' & \mathbb{L} & a & t & !m & n & p \\
\hline
\end{array}
\]

C.1.8 Discard Record

A discard record is logged each time a timer is explicitly stopped. The record includes the current value of the system logical clock $\mathbb{L}$, the identity of the owner $a$ of timer $t$, and the identity of the timer $t$ to stop.

\[
\begin{array}{|c|c|c|}
\hline
'd' & \mathbb{L} & a & t \\
\hline
\end{array}
\]

C.1.9 Transmission Record

A transmission record is logged each time a monitorable output message is generated. The record includes the current value of the system logical clock $\mathbb{L}$, the identity of the message being transmitted $m$, and the identity of the target actor $b$. 

\[
\begin{array}{|c|c|}
\hline
'\phi' & \mathbb{L} \\
\hline
\end{array}
\]
C.1.10 Commitment Record

A commitment record is logged each time an active actor reaches a commitment point. The record includes the current value of the system logical clock $L$, the commitment operation executed $r$, and the next conceptual state of the actor $q_{next}$.

```
's'  L  !m  b
```

C.1.11 Exception Record

An exception record is logged each time a run-time exception is generated. The record includes the current value of the system logical clock $L$, a machine specific context $c$, the current conceptual state $q$ of the active actor, and an exception code $e \in E$ (see Section 4.4.3).

```
'c'  L  r  q_{next}
```

```
'e'  L  c  q  e
```
Appendix D

COOL Specifications for Case Studies

D.1 Listings for Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Listings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vending machine example.</td>
<td>Listing D.1 starting on page 228.</td>
</tr>
<tr>
<td>Application DLOCK—Distributed lock protocol.</td>
<td>Listings D.7–D.10 starting on page 234.</td>
</tr>
</tbody>
</table>

Table D.1: COOL specifications for case studies.
Listing D.1: COOL specification for a vending machine—\( v \in \text{Vend}, u \in \text{User}, \) and \( m \in \text{Main} \).

\begin{verbatim}
actor class Vend {
  int x;
  User u;

Start:
  msg init () {
    becomes Select ;
  }

Select:
  msg select (int y) {
    u = (User) * ;
    x = y ; becomes VM0 ;
  }

VM0:
  msg coin (int y) from u {
    if (y == 5) becomes VM5 ;
    if (x == 'a') becomes VM10 ;
    dispense ('a') ⇒ u ; becomes Select ;
  }

VM5:
  msg coin (int y) from u {
    if (x == 'a') {
      dispense ('a') ⇒ u ; becomes Select ;
    }
    if (y == 5) becomes VM10 ;
    dispense ('b') ⇒ u ; becomes Select ;
  }

VM10:
  msg coin (int y) from u {
    dispense ('b') ⇒ u ; becomes Select ;
  }

Defaults:
  msg restart () from u {
    becomes Select ;
  }
}

actor class User {
  Timer t ;
  Vend v ;

Start:
  msg init (Vend x) {
    v = x ;
    t = trigger noselect () on 15 sec ;
    select ('b') ⇒ v ; coin (5) ⇒ v ;
    coin (5) ⇒ v ; coin (5) ⇒ v ;
    becomes Dispense ;
  }

Dispense:
  msg dispense (int x) from v {
    discard t ;
    terminate ;
  }

Defaults:
  msg noselect () from t {
    restart () ⇒ v ;
    becomes Start ;
  }

Satisfies:
  \( \phi_1 \triangleq \text{init} \sim \text{dispense} : 15 \text{ sec} \)
}

actor class Main {
  Start:
    msg init () {
      User u = new User ;
      Vend v = new Vend ;
      init (v) ⇒ u ;
      init () ⇒ v ;
      terminate ;
    }
}
\end{verbatim}
Listing D.2: Application DBUFF—specification for \( b \in \text{Buffer} \).

```plaintext
actor class Buffer {
    Producer p;
    Consumer c;
    const real \( \mu = 10.0 \);
    const real \( \sigma = 1.5 \);
    const int \( n = 20 \);
    fun void initQueue(int n);
    fun void enQueue(int x);
    fun int deQueue(void);
    fun bool isFullQ(void);
    fun bool isEmptyQ(void);

Start:
    msg init(Producer x, Consumer y) {
        p = x; c = y;
        initQueue(n);
        becomes Empty;
    }

Empty:
    msg put(int x) from p {
        enQueue(x);
        if (isFullQ() == true) becomes Full;
        else becomes PartFull;
    }
    msg get() from c {
        underflow() \Rightarrow c;
    }

Full:
    msg get() from c {
        out(deQueue()) \Rightarrow c;
        if (isEmptyQ() == true) becomes Empty;
        else becomes PartFull;
    }
}
```

\( \phi_1 \triangleq 0 \leq (\#? put - \#! overflow) - \#! out \leq n \)
\( \phi_2 \triangleq (\mu - \sigma) \leq \#! out - \#! out \leq (\mu + \sigma) \)
\( \phi_3 \triangleq 0 \leq \#? put - \#? put \leq (\mu \times (n - 1)) \)
\( \phi_4 \triangleq (\phi_1 \land \phi_2) \lor \phi_3 \)
Listing D.3: Application DBUFF—specification for $p \in \text{Producer}$, $c \in \text{Consumer}$, and $m \in \text{Main}$.

```plaintext
actor class Producer {
  Buffer b;
  Timer t;
  const int freq = 50;
  fun int Produce(void);
Start:
  msg init(Buffer x) {
    b = x;
    t = trigger gen() on freq;
    becomes Generate;
  }
Generate:
  msg gen() from t {
    put(Produce()) ⇒ b;
    t = trigger gen() on freq;
  }
  msg overflow(int x) from b {
    put(x) ⇒ b;
    t = trigger gen() on freq;
  }
  Defaults:
  msg stop() from owner {
    terminate;
  }
}
actor class Consumer {
  Buffer b;
  fun void Consume(int x);
Start:
  msg init(Buffer x) {
    b = x;
    get() ⇒ b;
    becomes Consume;
  }
Consume:
  msg out(int x) from b {
    Consume(x);
    get() ⇒ b;
  }
  msg underflow() from b {
    get() ⇒ b;
  }
  Defaults:
  msg stop() from owner {
    terminate;
  }
}
actor class Main {
Start:
  msg init() {
    Buffer b = new Buffer;
    Producer p = new Producer;
    Consumer c = new Consumer;
    init(p,c) ⇒ b;
    init(b) ⇒ p;
    init(b) ⇒ c;
    terminate;
  }
}
```
actor class Sender {
    Receiver r;
    User u;
    Timer t; // tick
    int nfts; // Next frame to send
    const int ackTimeout = 4;
    fun int inc(int x, int y);
    fun int dec(int x, int y);
    Start:
    msg init(Receiver x) {
        r = x;
        u = (User) *;
        nfts = 0;
        sapa() => u;
        becomes Tx;
    }
    Tx:
    msg nf(int x) from u {
        f(nfts,x) => r;
        t = trigger noack(nfts,x) on ackTimeout;
    }
    msg g(int seq) from r dprob 0.001 {
        if (nfts == seq) // ACK
            discard t;
            nfts = inc(nfts,MAXSEQ);
            sapa() => u;
        }
        else { ; } // NAK
    }
    msg noack(int seq, int x) from t {
        f(seq,x) => r;
        t = trigger noack(seq,x) on ackTimeout;
    }
}

actor class Receiver {
    Sender s;
    User u;
    int fe; // Frame expected
    int nd; // Next datum
    fun int inc(int x, int y);
    fun int dec(int x, int y);
    fun bool checkData(int x);
    Start:
    msg init(Sender x) {
        s = x;
        u = (User) *;
        fe = 0;
        nd = 0;
        becomes Rx;
    }
    Rx:
    msg f(int seq, int x) from s dprob 0.001 {
        if (fe == seq) {
            fe = inc(fe,MAXSEQ);
            if (checkData(x) == false)
                abort;
            sapb(x) => u;
            g(seq) => s;
        }
    }
Listing D.5: Application DABP—specification for \( r \in \text{Receiver} \) (continued) and \( u \in \text{User} \).

```plaintext
26    else {
27        g(\text{dec}(fe, \text{MAXSEQ})) \Rightarrow s;
28    }
29}
30
31  Defaults:
32  \text{msg stop()} \text{ from owner } {
33      \text{terminate;}
34  }
35
36  Satisfies:
37  \phi_3 \triangleq f(seq, x) \text{ assert}(fe == seq)
38  \phi_4 \triangleq 0 \leq \num{f - \text{maxFrames}} \leq \num{300 usec}
```

```plaintext
1  \text{actor class User {}
2      Receiver \ r;
3      Sender \ s;
4      int seqno, maxFrames = 102400;
5      int nd;
6      fun int getFrame();
7      fun void putFrame(int x);
8  }
9
10  \text{Start:}
11  \text{msg init(string role, Receiver x, Sender y) {}
12     if (role == "Sender") {
13         r = x; s = y;
14         init(r) \Rightarrow s;
15         becomes TestSender;
16     }
17     else {
18         r = x; s = y;
19         init(s) \Rightarrow r;
20         becomes TestReceiver;
21     }
22  }
```
Listing D.6: Application DABP—specification for \( m \in \text{Main}^* \), constants, and function definitions.

actor class Main_Sender {
  // This class bootstraps the sending node.
  User us = new User;
  Sender s = new Sender;

  Start:
  msg init () {
    Connect c = new Connect;
    connect( "UserS", us, "Sender", s, "Receiver" ) => c;
  }

  msg connected (actor x, actor y, actor r) {
    init( "Sender", r, s ) => us;
    terminate;
  }
}

actor class Main_Receiver {
  // This class bootstraps the receiving node.
  User ur = new User;
  Receiver r = new Receiver;

  Start:
  msg init () {
    Connect c = new Connect;
    connect( "UserR", ur, "Receiver", r, "Sender" ) => c;
  }

  msg connected (actor x, actor y, actor s) {
    init( "Receiver", r, s ) => ur;
    terminate;
  }
}

// Constants and function definitions
#define MAXSEQ 1
#define MAXCOLOR 7

fun int inc(int x, int y) { return (x + 1) % (y + 1); }
fun int dec(int x, int y) { return (x + y) % (y + 1); }
fun void putFrame(int x) { ; }
fun int getFrame () {
  int rc = nd;
  nd = inc(nd, MAXCOLOR);
  return rc;
}
fun bool checkData(int x) {
  if ((x != nd) || (chance(0.001) == true))
    return false;
  nd = inc(nd, MAXCOLOR);
  return true;
}
Listing D.7: Application DLOCK—specification for \( d \in \text{DLock} \).

```java
1 actor class Service;
2 actor class DLock {
3    Timer t; // \( t_{\text{D}} \)
4    Service s;
5    actor h, k;
6    int quanta;
7 Start:
8    msg init(int x) {
9        quanta = x;
10       h = k = none;
11       becomes Srv;
12    }
13 Srv:
14    msg lock(actor p) {
15        if (h == none) {
16            t = trigger noquantum() on quanta;
17            h = *;
18            s = new Service;
19            init(p) ⇒ s;
20            locked(s) ⇒ h;
21        }
22        else if (h == * ) {
23            locked_r(s) ⇒ *;
24        }
25        else {
26            reject() ⇒ *;
27        }
28    }
29 msg unlock() {
30        if (h == * ) {
31            discard t;
32            stop() ⇒ s;
33            unlocked() ⇒ h;
34            k = h;
35            h = none;
36        }
37    }
38 else if (k == * ) {
39        unlocked_r() ⇒ *;
40    }
41 else {
42        reject() ⇒ *;
43    }
44 msg noquantum() from t if (h != none) {
45            stop() ⇒ s;
46            unlocked() ⇒ h;
47            k = h;
48            h = none;
49    }
50 Defaults:
51    msg stop() from owner {
52        terminate;
53    }
54 Satisfies:
55    \( \phi_1 \triangleq 0 \leq \#\text{locked} - \#\text{unlocked} \leq 1 \)
56 }
```

```java
1 actor class Locker {
2    int \( n_1 \); // Duration of \( t \)
3    int \( n_2 \); // Maximum lock attempts
4    int n; // Current number of attempts
5    Timer t; // \( t_{\text{lock}} \), for bounded retransmission
6    DLock d; // Distributed lock
7    actor p; // User of protected service
8 Start:
9    msg init(int x, int y) from owner {
10        // Try for \( (n_1 \times n_2) \) milliseconds
11        \( n_1 = x; \)
12        \( n_2 = y; \)
13        becomes idle ;
14    }
```

Listing D.8: Application DLOCK—specification for \( l \in \text{Locker} \).

\begin{verbatim}
15 idle:
16  msg lock(actor x,actor y) { // Request lock
17      d = (DLock) x; p = y;
18      n = 0;
19      t = trigger noresponse(1) on n;
20      lock(p) ⇒ d;
21      becomes Locking;
22  }
23  msg unlocked_r() {
24      commit;
25  }
26 locking:
27  msg locked(actor s) dprob 0.01 { // Lock granted
28      discard t;
29      locked(s) ⇒ owner;
30      becomes Locked;
31  }
32  msg locked_r(actor s) dprob 0.01 {
33      // Message locked() can be lost
34      discard t;
35      locked(s) ⇒ owner;
36      becomes Locked;
37  }
38  msg reject() { // Lock in use.
39      commit;
40  }
41 locked:
42  msg unlock(actor x) { // Release lock
43      d = (DLock) x;
44      n = 0;
45      t = trigger noresponse(2) on n;
46      unlock() ⇒ d;
47      becomes Release;
48  }
49  msg unlocked() {
50      // Quantum expired — user loses lock
51      expired() ⇒ owner;
52      becomes idle;
53  }
54  msg locked_r(actor s) { // Duplicates of locked() exist
55      commit;
56  }
57 release:
58  msg unlocked() dprob 0.01 {
59      // Lock released — user released lock
60      discard t;
61      unlocked() ⇒ owner;
62      becomes idle;
63  }
64  msg unlock_r() dprob 0.01 {
65      // Message unlock() can be lost
66      discard t;
67      unlocked() ⇒ owner;
68      becomes idle;
69  }
70  msg reject() { // Lock busy.
71      commit;
72  }
73 defaults:
74  msg noresponse(int op) from t {
75      if (++n <= n2) {
76          t = trigger noresponse(op) on n;
77          if (op == 1) lock(p) ⇒ d;
78          else if (op == 2) unlock() ⇒ d;
79          commit;
80          apperror(op) ⇒ owner;
81          becomes idle;
82      }
83  }
\end{verbatim}
Listing D.9: Application DLOCK—specification for $u \in \text{User}$.

Satisfies:
\[
\begin{align*}
\phi_2 & \triangleq \#!\text{lock}, \phi_3 \triangleq \#!\text{unlock}, \phi_4 \triangleq \#?\text{locked} \\
\phi_5 & \triangleq \#?\text{locked}_{\_r}, \phi_6 \triangleq \#?\text{unlocked}, \phi_7 \triangleq \#?\text{unlocked}_{\_r}
\end{align*}
\]

1. **actor class User**
   2. `DLock d;`
   3. `Locker l;`
   4. `Service s;`
   5. `Proxy p;`
   6. `Timer t;` // treq
   7. `int nSessions;`
   8. `bool busy;`
   9. `const int mSessions = 256;`

10. **Start**:
   11. `msg init(DLock x, int y, int z) {`
   12.     `d = x;`
   13.     `nSessions = 0;`
   14.     `p = new Proxy;`
   15.     `l = new Locker;`
   16.     `init(y, z) ⇒ l;`
   17.     `t = trigger nextAttempt() on 50 msec;`
   18.     `busy = false;`
   19.     `becomes Run;`
   20. }

21. **Run**:
   22. `msg nextAttempt() from t {`
   23.     `if (busy == false) {`
   24.         `if (nSessions >= mSessions) {`
   25.             `stop() ⇒ l; stop() ⇒ p;`
   26.             `terminate;`
   27.         }
   28.         `// Request Lock`
   29.         `lock(d, p) ⇒ l;`
   30.         `busy = true;`
   31.     }
   32.     `t = trigger nextAttempt() on 50 msec;`
   33. }
   34. `msg locked(actor x) from l {`
   35.     `// lock granted`
   36.     `s = x;`
   37.     `start(s) ⇒ p;`
   38. }
   39. `msg complete() from p {`
   40.     `// service complete`
   41.     `unlock(d) ⇒ l;`
   42.     `end() ⇒ p;`
   43. }
   44. `msg unlocked() from l {`
   45.     `// lock released`
   46.     `nSessions += 1;`
   47.     `busy = false;`
   48. }
   49. `msg expired() from l {`
   50.     `// quantum expired — lock released`
   51.     `nSessions += 1;`
   52.     `reset() ⇒ p;`
   53.     `busy = false;`
   54. }

55. **Defaults**:
   56. `msg apperror(int x) {`
   57.     `busy = false;`
   58. }
   59. `msg stop() from owner {
   60.     `terminate;`
   61. }

62. **Satisfies**:
   63. `φ_8 \triangleq \#?\text{expired}
   64. `φ_9 \triangleq \#?\text{apperror}
Listing D.10: Application DLOCK—specification for \( m \in \text{Main}^* \).

```java
actor class Main_Node_One {
    // This class bootstraps the node with distributed service protected by distributed lock \( d \).
    DLock d = new DLock;
    const int quantum = 40;

    Start:
    msg init() {
        Connect c = new Connect;
        connect( "Server", d, "User1", "User2" ) ⇒ c;
    }

    msg connected(actor x, actor y, actor z) {
        init(quantum) ⇒ d;
        terminate;
    }
}

actor class Main_Node_Two {
    // This class bootstraps a user on Node 2.
    const int mAttempts = 40;
    const int mPeriod = 20;

    Start:
    msg init() {
        Connect c = new Connect;
        connect( "User2", self, "User1", "Server" ) ⇒ c;
    }

    msg connected(actor x, actor y, actor d) {
        User u;
        u = new User; init(d, mPeriod, mAttempt) ⇒ u;
    }
}

actor class Main_Node_Three {
    // This class bootstraps a user on Node 3.
    const int mAttempts = 40;
    const int mPeriod = 20;

    Start:
    msg init() {
        Connect c = new Connect;
        connect( "User2", self, "User1", "Server" ) ⇒ c;
    }

    msg connected(actor x, actor y, actor d) {
        User u;
        u = new User; init(d, mPeriod, mAttempt) ⇒ u;
    }
}
```
D.2 Application DLOCK, Simulation Results

Table D.2: Enumeration characteristics of application DLOCK.

<table>
<thead>
<tr>
<th>Enumeration</th>
<th>Class</th>
<th>Locker l₁</th>
<th>Locker l₂</th>
<th>Locker l₃</th>
<th>Locker l₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node 2, Trial 1, n = 256</td>
<td>User</td>
<td>19</td>
<td>15</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>User</td>
<td>13</td>
<td>20</td>
<td>17</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>Locker</td>
<td>3478</td>
<td>3570</td>
<td>3440</td>
<td>3630</td>
</tr>
<tr>
<td></td>
<td>Locker</td>
<td>240</td>
<td>244</td>
<td>241</td>
<td>248</td>
</tr>
<tr>
<td></td>
<td>Locker</td>
<td>254</td>
<td>254</td>
<td>253</td>
<td>251</td>
</tr>
<tr>
<td></td>
<td>Locker</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Locker</td>
<td>253</td>
<td>253</td>
<td>251</td>
<td>249</td>
</tr>
<tr>
<td></td>
<td>User</td>
<td>9</td>
<td>25</td>
<td>14</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>User</td>
<td>18</td>
<td>16</td>
<td>14</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>Locker</td>
<td>3764</td>
<td>3724</td>
<td>3680</td>
<td>3826</td>
</tr>
<tr>
<td></td>
<td>Locker</td>
<td>250</td>
<td>234</td>
<td>242</td>
<td>239</td>
</tr>
<tr>
<td></td>
<td>Locker</td>
<td>254</td>
<td>249</td>
<td>252</td>
<td>251</td>
</tr>
<tr>
<td></td>
<td>Locker</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Locker</td>
<td>253</td>
<td>253</td>
<td>256</td>
<td>254</td>
</tr>
<tr>
<td></td>
<td>User</td>
<td>13</td>
<td>24</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>User</td>
<td>14</td>
<td>17</td>
<td>18</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>Locker</td>
<td>3614</td>
<td>3596</td>
<td>3624</td>
<td>3690</td>
</tr>
<tr>
<td></td>
<td>Locker</td>
<td>245</td>
<td>236</td>
<td>243</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>Locker</td>
<td>253</td>
<td>251</td>
<td>251</td>
<td>255</td>
</tr>
<tr>
<td></td>
<td>Locker</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Locker</td>
<td>254</td>
<td>252</td>
<td>255</td>
<td>252</td>
</tr>
<tr>
<td></td>
<td>Locker</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Locker</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

continued on next page
<table>
<thead>
<tr>
<th>Enumeration</th>
<th>Class</th>
<th>Locker 1</th>
<th>Locker 2</th>
<th>Locker 3</th>
<th>Locker 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>#drop(locked_r)</td>
<td>Locker</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>#drop(unlocked)</td>
<td>Locker</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>#drop(unlocked_r)</td>
<td>Locker</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Node 2, Trial 4, \( n = 256 \)

| #? expired            | User   | 15       | 15       | 18       | 16       |
| #? apperror           | User   | 17       | 9        | 16       | 18       |
| #! lock               | Locker | 3663     | 3234     | 3590     | 3625     |
| #! unlock             | Locker | 245      | 244      | 242      | 243      |
| #! locked             | Locker | 255      | 255      | 253      | 254      |
| #? locked_r           | Locker | 1        | 1        | 3        | 2        |
| #? unlocked           | Locker | 252      | 253      | 252      | 253      |
| #? unlocked_r         | Locker | 4        | 3        | 4        | 3        |
| #drop(locked)         | Locker | 1        | 1        | 3        | 2        |
| #drop(locked_r)       | Locker | 0        | 0        | 0        | 0        |
| #drop(unlocked)       | Locker | 1        | 1        | 3        | 2        |
| #drop(unlocked_r)     | Locker | 0        | 0        | 0        | 0        |

Node 2, Trial 5, \( n = 256 \)

| #? expired            | User   | 21       | 24       | 16       | 23       |
| #? apperror           | User   | 12       | 16       | 18       | 18       |
| #! lock               | Locker | 3297     | 3379     | 3666     | 3453     |
| #! unlock             | Locker | 239      | 234      | 245      | 237      |
| #! locked             | Locker | 253      | 255      | 256      | 252      |
| #? locked_r           | Locker | 3        | 1        | 0        | 4        |
| #? unlocked           | Locker | 252      | 254      | 251      | 252      |
| #? unlocked_r         | Locker | 4        | 2        | 5        | 5        |
| #drop(locked)         | Locker | 3        | 1        | 0        | 4        |
| #drop(locked_r)       | Locker | 0        | 0        | 0        | 0        |
| #drop(unlocked)       | Locker | 4        | 2        | 5        | 4        |
| #drop(unlocked_r)     | Locker | 0        | 0        | 0        | 0        |

Node 3, Trial 1, \( n = 256 \)

| #? expired            | User   | 51       | 34       | 39       | 34       |
| #? apperror           | User   | 18       | 23       | 20       | 25       |
| #! lock               | Locker | 3570     | 3587     | 3572     | 3637     |
| #! unlock             | Locker | 210      | 225      | 220      | 223      |
| #! locked             | Locker | 251      | 253      | 253      | 251      |
| #? locked_r           | Locker | 5        | 3        | 3        | 5        |
| #? unlocked           | Locker | 252      | 253      | 253      | 255      |
| #? unlocked_r         | Locker | 5        | 4        | 5        | 1        |
| #drop(locked)         | Locker | 5        | 3        | 3        | 5        |
| #drop(locked_r)       | Locker | 0        | 0        | 0        | 0        |

continued on next page
### Node 3, Trial 2, $n = 256$

<table>
<thead>
<tr>
<th>Enumeration</th>
<th>Class</th>
<th>Locker 1</th>
<th>Locker 2</th>
<th>Locker 3</th>
<th>Locker 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>#drop(unlocked)</td>
<td>Locker</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>#drop(unlocked_r)</td>
<td>Locker</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Node 3, Trial 3, $n = 256$

<table>
<thead>
<tr>
<th>Enumeration</th>
<th>Class</th>
<th>Locker 1</th>
<th>Locker 2</th>
<th>Locker 3</th>
<th>Locker 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>#? expired</td>
<td>User</td>
<td>47</td>
<td>50</td>
<td>48</td>
<td>44</td>
</tr>
<tr>
<td>#? apperror</td>
<td>User</td>
<td>16</td>
<td>10</td>
<td>17</td>
<td>20</td>
</tr>
<tr>
<td>#! lock</td>
<td>Locker</td>
<td>3090</td>
<td>3155</td>
<td>3638</td>
<td>3684</td>
</tr>
<tr>
<td>#! unlock</td>
<td>Locker</td>
<td>210</td>
<td>210</td>
<td>213</td>
<td>216</td>
</tr>
<tr>
<td>#? locked</td>
<td>Locker</td>
<td>256</td>
<td>253</td>
<td>252</td>
<td>251</td>
</tr>
<tr>
<td>#? locked_r</td>
<td>Locker</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>#? unlocked</td>
<td>Locker</td>
<td>255</td>
<td>252</td>
<td>251</td>
<td>252</td>
</tr>
<tr>
<td>#? unlocked_r</td>
<td>Locker</td>
<td>3</td>
<td>7</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>#drop(locked)</td>
<td>Locker</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>#drop(locked_r)</td>
<td>Locker</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>#drop(unlocked)</td>
<td>Locker</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>#drop(unlocked_r)</td>
<td>Locker</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Node 3, Trial 4, $n = 256$

<table>
<thead>
<tr>
<th>Enumeration</th>
<th>Class</th>
<th>Locker 1</th>
<th>Locker 2</th>
<th>Locker 3</th>
<th>Locker 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>#? expired</td>
<td>User</td>
<td>44</td>
<td>34</td>
<td>46</td>
<td>42</td>
</tr>
<tr>
<td>#? apperror</td>
<td>User</td>
<td>8</td>
<td>12</td>
<td>11</td>
<td>22</td>
</tr>
<tr>
<td>#! lock</td>
<td>Locker</td>
<td>3298</td>
<td>3261</td>
<td>3097</td>
<td>3532</td>
</tr>
<tr>
<td>#! unlock</td>
<td>Locker</td>
<td>212</td>
<td>224</td>
<td>210</td>
<td>217</td>
</tr>
<tr>
<td>#? locked</td>
<td>Locker</td>
<td>253</td>
<td>251</td>
<td>252</td>
<td>254</td>
</tr>
<tr>
<td>#? locked_r</td>
<td>Locker</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>#? unlocked</td>
<td>Locker</td>
<td>256</td>
<td>254</td>
<td>256</td>
<td>253</td>
</tr>
<tr>
<td>#? unlocked_r</td>
<td>Locker</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>#drop(locked)</td>
<td>Locker</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>#drop(locked_r)</td>
<td>Locker</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>#drop(unlocked)</td>
<td>Locker</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>#drop(unlocked_r)</td>
<td>Locker</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### continued on next page
There are eight service users and associated lock coordinators. The results for service user $u_1$ and coordination $l_1$ on node 2 are reported in Section 7.3.2.
Index of Symbols

Common Symbols

\( X^* \) all possible sequences of elements from domain \( X \) \hspace{1cm} 31
\( \mathcal{P}(K) \) the power set of \( K \), the set of all finite subsets of \( K \) \hspace{1cm} 31
\( \mathbb{N} \) the natural numbers, 0, 1, 2, \( \cdots \) \hspace{1cm} 30

Chapter 1: Introduction

\( E \) an executable implementation \hspace{1cm} 5
\( M \) a COOL specification \hspace{1cm} 5
\( M' \) a source implementation, VM instructions \hspace{1cm} 5
\( \Phi \) the set of checkable properties \hspace{1cm} 5
\( \Phi' \) the set of canonical checkable properties \hspace{1cm} 6

Chapter 2: Actor Systems

\( A \) the set of all possible actor identities, \( a \in A \) \hspace{1cm} 28
\( B \) the set of all possible actor behaviours, \( b \in B \) \hspace{1cm} 30
\( C \) a timed actor configuration, \( \langle \mathcal{K}, J, X \rangle \) \hspace{1cm} 32
\( D \) the name of an actor definition \hspace{1cm} 36
\( J \) the set of all possible timer instances, \( J = T \times Q \times S'; \ j \in J \) \hspace{1cm} 29
\( J^+ \) the set of fresh timers created by an activated actor \hspace{1cm} 31
\( J^- \) the set of timers discarded by an activated actor \hspace{1cm} 31
\( K \) the set of all possible actor instances; \( K = A \times Q \times S; \ k \in K \) \hspace{1cm} 28
\( K^+ \) the set of fresh actors created by an activated actor \hspace{1cm} 31
\( Q \) the set of all possible conceptual states, \( q \in Q \) \hspace{1cm} 28
\( R \) the set of all possible commitments, \( r \in R \)  
\( S \) the set of all possible local states, \( s \in S \)  
\( S' \subseteq S \) the set of all possible timer local states, \( s \in S' \)  
\( T \) the set of all possible timer identities, \( t \in T \)  
\( V \) the set of all possible communications, \( v \in V \)  
\( X \) the set of all possible messages, \( \alpha, \beta \in X \)  
\( X^+ \) the sequence of messages generated by an activated actor  
\( \mathbb{T} \) the local master-clock  
\( I \) the intentions list, records operations executed during activation  
\( J \subseteq J \) the set of timer instances in configuration \( C \)  
\( \mathcal{K} \subseteq \mathcal{K} \) the set of actor instances in configuration \( C \)  
\( \mathcal{X} \subseteq \mathcal{X} \) the set of pending messages in configuration \( C \)  
\( \Sigma \) the set of all possible input message names  
\( \Sigma_D \) the set of input messages actor instance \( a \in D \) can receive  
\( \beta \) timeout message, \( \beta = (t, a, \sigma, v) \)  
\( \sigma \in \Sigma \) an unspecified input message  
\( \omega \) the activation time (\( \mathbb{T} \)) of actor \( a \)  
\( \sqrt{.} \) a special action which marks the progress of time

Chapter 3: An Asynchronous Actor Algebra

\textit{Act} the action set accepted by actor and timer terms in an ACube expression \( \mathcal{E} \)  
\( D, D_C \) an actor class definition  
\( P \) a message handler  
\( A, B \) actor terms  
\( C \) the set of actor class names, \( C \in C \)  
\( \mathcal{E} \) the set of ACube expressions  
\( M \) a message term  
\( T \) a timer term
κ a term on the intentions list 57

the marshalling of parameters \( \theta_m \) is denoted by

\[
\theta_m, \theta_u \left\{ v_1 \leftarrow [e_1], v_2 \leftarrow [e_2], \cdots, v_n \leftarrow [e_n] \right\}, \text{ while the unmarshalling of parameters} \\
\theta_u \text{ is denoted by } \left\{ x_1 \leftarrow v_1, x_2 \leftarrow v_2, \cdots, x_n \leftarrow v_n \right\}
\]

τ the silent action 53

\( a_q(D)_s \) a ready actor; one waiting for its next message 50

\( a_q[D]_{s \sigma v} \) an active actor; one currently executing a message handler 50

\( t_a(D)_{n,p}^\beta \) a ready timer; one waiting for a \( \sqrt{\cdot} \), trigger, or discard operation 50

\( t_a[D]_{n,p}^\beta \) an active timer; one processing a \( \sqrt{\cdot} \), trigger or discard operation 51

Chapter 4: A Virtual Machine for a Timed Actor Language

\( L \) a scalar logical clock managed by the run-time kernel, partially orders system events 75

\( T \) the nodes master-clock, generates the \( \sqrt{\cdot} \) action which updates timers 73

\( B_N \) the message buffer pool managed by the node and VM, storage for \( R \) and \( J \) 75

\( B_V \) the message buffer pool managed by VM, storage for \( X_L, X_I, X_O, \) and \( J \) 75

\( I \) the intentions list, contains the history of an actors execution; atomicity support 75

\( J \) the timers list, the set of ready and active timers in the local configuration 75

\( K \) the actors list, the set of ready and active actors in the local configuration 75

\( R \) the system receive buffer, distributed messages arrive at the VM through \( R \) 74

\( S \) the state list, \( S = S_K \cup S_J \) 75

\( S_J \) the state list for actors, accommodates the local state of actors in \( K \) 75

\( S_K \) the state list for timers, accommodates the local state of timers in \( J \) 75

\( T \) the system transmit buffer, distributed messages depart the VM through \( T \) 74

\( X \) the sequence of pending messages, \( X = X_L \cup X_I \cup X_O \) 75

\( X_I \) the sequence of inbound messages, messages originating from another node 75

\( X_L \) the sequence of local messages, messages originating and processed by the local configuration 75

\( X_O \) the sequence of outbound messages, messages originating in the local configuration but processed on another node 75
Chapter 5: COOL—A Timed Actor Language

$\Sigma_C$ the set of messages actor $a \in C$ may receive 111

$\Sigma_{actor}$ union sigma; the set of messages actor $a$ may output 112

Chapter 6: Validating COOL Specifications

$Y$ a finite set of event types, $y \in Y$ 143

$z, z_1, z_2$ a communications event, message reception $?m$ or message transmission $!m$ 130

#$z$ an enumeration event 133

$'z$ a timed event $'z$ occurs in the current activation 135

"$z$ a timed event "$z$ occurs in a past activation 135

$z_1 \rightsquigarrow z_2$ a follows event, the time between two causally related communication events 132

$\mathbb{Q}$ the set of rational numbers: $\{a/b | a, b \in \mathbb{Z}, b \neq 0\}$ 136

$\Gamma$ a finite set of enumeration, $\gamma \in \Gamma$ where $\gamma = \#z \mid \gamma_1 \text{ op } \gamma_2$ 133

$\Phi(\Gamma)$ a finite set of enumeration constraints 133

$\rho \in \Phi(\Gamma)$ an enumeration constraint, $\rho ::= \gamma \mid \gamma \leq e \mid e \leq \gamma \mid \neg \rho \mid \rho_1 \land \rho_2 \mid \rho_1 \lor \rho_2$ 133

$\Delta$ a finite set of timed intervals, $\delta \in \Delta$ where $\delta = 'z_1 - ''z_2$ 136

$\Phi(\Delta)$ a finite set of timing constraints 136

$\lambda \in \Phi(\Delta)$ a timing constraint, $\lambda ::= \delta \mid \delta \leq e \mid e \leq \delta \mid \neg \lambda \mid \lambda_1 \land \lambda_2 \mid \lambda_1 \lor \lambda_2$ 136

$\phi, \phi_1, \phi_2$ a checkable property, checkable property definition, or checkable property name 131

Chapter B: Details of a Virtual Machine Implementation

$J \& \subseteq Rf$ the set of all possible references to timer instances, $o \in J \&$ 213

$K \& \subseteq Rf$ the set of all possible references to actor instances, $o \in K \&$ 213

$L$ an ordered or unordered list, $l \in L$ 213

$N \subseteq \mathbb{N}$ the set of node identities, $n \in N$ 209

$P$ a timer type, $p \in \{\text{oneshot, periodic}\}$ 209

$R$ a commitment action, $r \in \{\text{becomes, abort, terminate}\}$ 209

$Rf$ a reference to an object instance, $o \in Rf$ 209

$U$ a configuration update action, $u \in \{\text{commit, rollback}\}$ 209
$X \& \subseteq Rf$ the set of all possible message references; $o \in X \&$

$\mathbb{B}$ the boolean values, $b \in \{false, true\}$

$\mathbb{C} \subseteq \mathbb{N}$ a class identity, $c \in \mathbb{C}$

$\mathbb{M}$ the set of message identities, $d \in \mathbb{M}$

$\mathbb{V}$ describes a function with no parameters or no return value
VITA

Surname: O’Connell

Given Names: Gordon Wayne

Place of Birth: New Westminster, British Columbia, Canada

Date of Birth: October 6, 1953

Educational Institutions Attended:

University of Victoria 1998 to 2004
University of Victoria 1991 to 1997
University of Victoria 1988 to 1991
University of the Pacific 1977 to 1979
University of Victoria 1973 to 1976

Degrees Awarded:

M. Sc. University of Victoria 1997
B. Sc. University of Victoria 1991
B. Sc.(Honours) University of Victoria 1976

Publications:


UNIVERSITY OF VICTORIA PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my dissertation to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this dissertation for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this dissertation for financial gain shall not be allowed without my written permission.

Title of Dissertation:

A Concurrent Object Coordination Language:

Semantics and Applications

Author:

Gordon Wayne O’Connell
December 7, 2004