Towards On-line Logistics: the LinC Interaction Modeling Language

H.J. Honig
Towards On-line Logistics:

the LinC Interaction Modeling Language
Towards On-line Logistics:
the LinC Interaction Modeling Language

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft

op gezag van de Rector Magnificus prof. dr. ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op 15 november 2004 om 15:30 uur

door Hendrik Jacob HONIG,
elektrotechnisch ingenieur,
geboren te Gouda
Dit proefschrift is goedgekeurd door de promotoren:
Prof. dr. H.G. Sol en prof. dr. ir. G. Lodewijks

Samenstelling promotiecommissie:
Rector Magnificus, voorzitter
Prof. dr. H.G. Sol, Technische Universiteit Delft, promoter
Prof. dr. ir. G. Lodewijks, Technische Universiteit Delft, promoter
Prof. dr. A. van Deursen, Technische Universiteit Delft, CWI
Prof. dr. ir. J.J.M. Evers, emeritus hoogleraar Technische Universiteit Delft
Prof. dr. J.A.E.E. van Nuen, Erasmus Universiteit Rotterdam
Prof. dr. ir. H.J. Sips, Technische Universiteit Delft
Prof. dr. ir. A. Verbraeck, University of Maryland, Technische Universiteit Delft

Published and distributed by: DUP Science

DUP Science is an imprint of
Delft University Press
P.O.Box 98
2600 MG Delft
The Netherlands
Telephone: +31 15 27 85 678
Telefax: +31 15 27 85 706
E-mail: info@library.tudelft.nl

ISBN 90-407-2544-6

Keywords: on-line logistics, distributed objects, interaction contracts

Copyright © 2004 by H.J. Honig

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission from the publisher: Delft University Press.

Printed in The Netherlands
Preface

The thesis that you find before you is the result of my 3-year long cooperation with Joseph Evers, the originator and driving force behind AgileFrames, a platform and framework for distributed logistics. Like with many fields that are deeply rooted in long time practice and experience, logistics has quickly adopted computers and networks to perform its tasks and make them easier than before to handle. But also like with many other fields, this does not necessarily mean that the new means are used to their full potential. Doing so often requires a change of view, a new paradigm. AgileFrames is such a new paradigm, and my own contribution is to define one specific part of it: the logistic contracting layer. Let’s explain in short what it is all about; the rest of this thesis will provide the details.

New Logistics

New logistics [ELL99, Eve00] is a field that spans the entire globe. It is involved with actors performing physical handling of goods or materials; these actors (which may reside in the same computer or in systems that are 10,000 kilometers apart) communicate with each other to coordinate the handling process. The only requirement is that actors are connected by a network (most likely the Internet), and that they adhere to a certain standard to communicate and get something done at different locations. Like many other complex systems (e.g. society itself), logistic systems are evolving to networks of interacting actors that together accomplish the system’s goals. The benefits of actor networking can be compared to those of human networking: it provides a latent structure, part of which can be activated whenever a certain activity could benefit from an actor’s facilities, capabilities, or connections to further actors. The effect is that the network can be used quickly and efficiently to perform any given task, that the available resources are used effectively, and that it is possible to catch opportunities when they occur without the need to first build the necessary infrastructure. As is the case with human networking, actor networking requires an actor to know the social codes, conventions, and standards of its possible partners. The interactions have to conform to these standards in order to be effective, and violations of the standards will cause failure of the interaction or (possibly permanent) rejection of the actor from the network. Whereas in human communication these standards have grown in after years of experimenting and learning, actor networks require their explicit definition. The need for a precise description of the conventions guiding the communication between actors has been the main reason for the design of the LiNC platform and language.
Notes

Since part II is intended to define the LINC language as well as to document its design process, it contains a lot of information that may be less relevant to average users of the language. A special language reference guide, from which the rationale paragraphs have been omitted, is available separately; contact the author for more information. A final note is concerned with the pronunciation of ‘LINC’: the word is supposed to sound as ‘link’. The resulting auditory confusion is intentional, as it allows the word to be used not only to denote the language, but also the individual contracts that connect logistic actors.

Acknowledgments

The language defined in this thesis was not designed from scratch. Instead, it took its form by discussing and digesting several earlier versions of the language. Glimpses of those earlier versions can be found in a range of articles about AgileFrames and the use of logistic contracting [ELL99, ELL00, Eve00] and from a prototype implementation of a (non-distributed) compiler for the contracting language described in those publications. As the reader will notice, the design has undergone a lot of changes as the early ideas matured and the case studies were providing important new insights. Nevertheless, the key idea of using logistic contracts has always stayed intact, and remained the cornerstone of this research. A similar remark applies to the second case study (‘Cross Docking’). The overall architecture of this case was designed by Joseph Evers. To be fair to myself, it still took quite some time and lots of discussion before I arrived at the set of contracts that is presented in chapter 7.

As the field of logistics engineering was totally new to me, I am very much indebted to Joseph Evers and David Lindeijer. Most of the ideas presented in this thesis are rooted in Joseph’s visions of the field of logistics and its future. His ideas about virtual logistic enterprises and especially those about the modeling of communication (interaction) were very close and sometimes ahead of what was happening in other fields (e.g. e-commerce, work-flow systems, and coordination languages). It was David who introduced me to the world of RMI and Jini, and who already had some fairly detailed ideas of how these software architectures could be applied to logistics engineering. Without Joseph and David, I would not have been able to design the LINC language; I therefore thank them for their insights, and for the many discussions we had about these and other topics; it has been a pleasure.

Of course I would also like to thank my promotors, Prof. Dr. Henk Sol and Prof. Dr. Gabriel Lodewijks, for their valuable comments on the successive drafts of this thesis. Their comments have helped me to fill the gaps and to improve the overall structure. A special word of appreciation is for Ruud Sommerhalder, who had no direct interest in this research but was always available to discuss the topics I was writing about. As in previous projects in which I have worked with him, our discussions helped to shape my thoughts and provoked a lot of new ideas. Some of these I had time to pursue, while others are still at the stage of short notes; I hope I will be able to elaborate these at some later day. Ruud, I owe you!
Contents

I On-line logistics: WHY and HOWTO 1

1 Introduction 5
  1.1 The challenge of on-line logistics 5
  1.2 Enterprise Resource Planning 6
  1.3 The need for a new paradigm 7
  1.4 New Logistics 9
  1.5 Research outline 11
  1.6 Related developments 12
  1.7 Organization of this thesis 15

2 Context of our research 17
  2.1 AgileFrames 17
  2.2 FoRcEs 19
  2.3 TrAcEs 20
  2.4 LinC 20

3 Logistic contracting 23
  3.1 Actor Interaction Modeling (AIM) 23
    3.1.1 The role of service contracts 24
    3.1.2 The role of logistic actors 24
    3.1.3 The design method 25
    3.1.4 The language: LinC 26
  3.2 Logistic actors 27
    3.2.1 The LinC virtual world 28
3.2.3 Basic actor capabilities ........................................... 29
3.2.4 Delegation and subcontracting ................................ 29

3.3 Logistic contracts ..................................................... 30
  3.3.1 Service contracts ............................................... 33
  3.3.2 Why service contracts are not programs ................. 35
  3.3.3 Life cycle of service contracts .......................... 36

3.4 The Logistic Contract Space ....................................... 37

4 The design of a contracting language ............................ 39
  4.1 Requirements for the LinC design ............................ 39
    4.1.1 Domain-specific requirements ......................... 40
    4.1.2 General language design requirements ............. 42
  4.2 Requirements for a LinC implementation ................. 43

II The LinC Language ...................................................... 45

5 The LinC Language Specification ................................ 49
  5.1 Notation .......................................................... 49
  5.2 Keywords ........................................................ 50
  5.3 Names ............................................................ 50
  5.4 Types ............................................................. 51
  5.5 2-Party Service Contracts .................................... 53
  5.6 The Service Description ....................................... 54
  5.7 The Service Actors .............................................. 56
  5.8 The Service Objects ............................................ 58
  5.9 The Service Parameters ....................................... 60
  5.10 The Service Indexes .......................................... 63
  5.11 The Service Variables ....................................... 63
  5.12 The Service Tasks ............................................. 65
7 Case Study: Cross Docking

7.1 Operational aspects .................................. 155
7.2 Design criteria ........................................ 156
7.3 The service actors ...................................... 158
7.4 The service contracts ................................... 166
  7.4.1 palletIntake ...................................... 166
  7.4.2 truckUnloading .................................... 175
  7.4.3 multiplePalletUnload .............................. 178
  7.4.4 singlePalletUnload .............................. 181
  7.4.5 palletStacking ................................... 183
  7.4.6 palletStoring ..................................... 185
  7.4.7 transferToStack .................................. 187
  7.4.8 moveToStack ..................................... 190
  7.4.9 palletOutlet ....................................... 192
  7.4.10 palletUnstacking ................................ 196
  7.4.11 palletRetrieving ................................. 198
  7.4.12 transferToPlatform .............................. 199
  7.4.13 moveToPlatform .................................. 199
  7.4.14 truckLoading ..................................... 199
  7.4.15 multiplePalletLoad .............................. 203
  7.4.16 singlePalletLoad ................................ 203
  7.4.17 iptBrokerage .................................... 206
Contents

7.4.18  iptRental1 .................................................. 206
7.4.19  iptRental2 .................................................. 209
7.4.20  iptRental3 .................................................. 211
7.4.21  iptRental4 .................................................. 211

IV Implementing the LinC platform ........................... 215

8 Distributed computing ........................................... 219
  8.1 Evolution of distributed systems ........................... 220
  8.2 Interaction mechanisms in distributed systems ............. 221
    8.2.1 Message based interaction .............................. 221
    8.2.2 Interaction through a shared data space ................ 223
    8.2.3 Remote procedure calls .................................. 223
    8.2.4 Distributed Objects .................................... 224
    8.2.5 P2P systems ............................................. 225
  8.3 Local/Remote transparency of RPC/RMI systems ............ 226
    8.3.1 Syntactic differences ................................... 226
    8.3.2 Semantic differences ................................... 227
  8.4 Interface Definition Languages ............................. 228
  8.5 Partial failure of distributed systems .................... 230
  8.6 Remote Object Discovery ................................... 230
    8.6.1 Naming and directory services ........................ 231
    8.6.2 Intentional naming ..................................... 231
    8.6.3 Peer discovery .......................................... 232
  8.7 The preferred platform for LinC .......................... 233
    8.7.1 Messages and/or methods? ............................. 233
    8.7.2 Language-centric or language independent? ............ 235
    8.7.3 Support for agile systems ............................. 235

9 The LinC platform architecture ............................... 237
9.1 The Logistic Contract Space .......................... 237
   9.1.1 Naming conventions .......................... 239
   9.1.2 Using the lookup service ...................... 239
   9.1.3 Fully and partially implemented services ...... 240
   9.1.4 Abstract versus concrete contracts .......... 241

9.2 Contract execution .................................. 241
   9.2.1 Contract instantiation ......................... 242
   9.2.2 Protocol verification .......................... 244
   9.2.3 Contract termination ......................... 246
   9.2.4 Execution journaling ......................... 246

9.3 Basic LinC actor capabilities ...................... 248
   9.3.1 Ability to inspect the journal tree .......... 249
   9.3.2 Ability to survive crashes .................. 249
   9.3.3 Actor replication ................................ 250
   9.3.4 Actor authentication .......................... 251

9.4 LinC tool support .................................. 251
   9.4.1 The LinC contract compiler .................... 252
   9.4.2 The service browser ............................ 252
   9.4.3 The actor browser .............................. 253
   9.4.4 The contract broker ............................ 253
   9.4.5 The journal tree browser ...................... 254

V Epilogue ............................................. 257

10 Evaluation of the design ............................. 261
   10.1 Using the LinC language ......................... 261
      10.1.1 Completeness of the description ............. 261
      10.1.2 Adequacy of the description ................ 262
      10.1.3 Contract verification ........................ 264
10.1.4 Ability to handle exceptional situations .......................... 265
10.1.5 Ability to escape from the contract ............................... 265
10.1.6 Structured logging .................................................. 266
10.2 Using the LinC implementation ...................................... 266
10.2.1 Contract rehearsal .................................................. 266
10.2.2 Contacting partners ................................................ 266
10.2.3 Standardization ..................................................... 267
10.2.4 Security .............................................................. 267
10.3 Concluding remarks ................................................... 267

11 Observations: the language and its use ............................. 269
11.1 Additions and changes motivated by the case studies ............ 269
11.1.1 The number of service objects .................................... 269
11.1.2 Service contract parameters ...................................... 270
11.1.3 Default values for service objects/parameters .................. 270
11.1.4 Delegation .......................................................... 271
11.1.5 Nested concurrent constructs ..................................... 272
11.1.6 Process error ....................................................... 272
11.1.7 Subcontracting inside multiple constructs ...................... 273
11.1.8 Exception handling ............................................... 273
11.1.9 Timing requirements ............................................... 274
11.1.10 Assertions/updates ................................................ 275
11.1.11 Premature contract termination .................................. 275
11.2 Open issues regarding the LinC language ......................... 276
11.2.1 Multiple indexes and exit constraints ............................ 276
11.2.2 Pre- and postconditions on service objects .................... 276
11.2.3 Pre- and postconditions on sub-contracts ...................... 277
11.2.4 Transactions ....................................................... 277
11.2.5 Actor restrictions on sub-contracts .............................. 278
11.2.6 Separation of instantiation and execution ........................................... 278
11.2.7 Concurrent use of shared resources ..................................................... 279
11.3 Using the LinC language ........................................................................ 280

A Introduction to Java RMI and Jini ............................................................. 283
   A.1 Introduction to Java RMI ...................................................................... 283
   A.2 Introduction to Jini ............................................................................. 284
      A.2.1 The lookup service ...................................................................... 286
      A.2.2 A simple Jini application .............................................................. 287

B Glossary of terms ...................................................................................... 291
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>AgileFrames</td>
</tr>
<tr>
<td>3.1</td>
<td>A provider can delegate parts of its service to subcontractors</td>
</tr>
<tr>
<td>3.2</td>
<td>Sample Logistic Contract Space</td>
</tr>
<tr>
<td>5.1</td>
<td>A full journal tree</td>
</tr>
<tr>
<td>5.2</td>
<td>Full journal tree with exception</td>
</tr>
<tr>
<td>5.3</td>
<td>Journal tree with try nodes suppressed</td>
</tr>
<tr>
<td>5.4</td>
<td>Journal tree with concurrent sequences</td>
</tr>
<tr>
<td>5.5</td>
<td>Journal tree with flattened concurrent sequences</td>
</tr>
<tr>
<td>6.1</td>
<td>Diagram of the value transport service</td>
</tr>
<tr>
<td>6.2</td>
<td>valueTransferService (part 1 of 2)</td>
</tr>
<tr>
<td>6.3</td>
<td>valueTransferService (part 2 of 2)</td>
</tr>
<tr>
<td>6.4</td>
<td>safeRental</td>
</tr>
<tr>
<td>6.5</td>
<td>multiplePrepareGetService</td>
</tr>
<tr>
<td>6.6</td>
<td>singlePrepareGetService</td>
</tr>
<tr>
<td>6.7</td>
<td>vanService (part 1 of 2)</td>
</tr>
<tr>
<td>6.8</td>
<td>vanService (part 2 of 2)</td>
</tr>
<tr>
<td>6.9</td>
<td>getService</td>
</tr>
<tr>
<td>7.1</td>
<td>Example Cross-Docking station</td>
</tr>
<tr>
<td>7.2</td>
<td>Example Cross-Docking station</td>
</tr>
<tr>
<td>7.3</td>
<td>Cross Docking: unloading (part 1 of 3)</td>
</tr>
<tr>
<td>7.4</td>
<td>Cross Docking: unloading (part 2 of 3)</td>
</tr>
<tr>
<td>7.5</td>
<td>Cross Docking: unloading (part 3 of 3)</td>
</tr>
<tr>
<td>FIGURE</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>7.6 Cross Docking: loading (part 1 of 3)</td>
<td>160</td>
</tr>
<tr>
<td>7.7 Cross Docking: loading (part 2 of 3)</td>
<td>161</td>
</tr>
<tr>
<td>7.8 Cross Docking: loading (part 3 of 3)</td>
<td>161</td>
</tr>
<tr>
<td>7.9 palletIntake (1 of 3)</td>
<td>167</td>
</tr>
<tr>
<td>7.10 palletIntake (2 of 3)</td>
<td>168</td>
</tr>
<tr>
<td>7.11 palletIntake (3 of 3)</td>
<td>169</td>
</tr>
<tr>
<td>7.12 truckUnloading</td>
<td>176</td>
</tr>
<tr>
<td>7.13 multiplePalletUnload</td>
<td>179</td>
</tr>
<tr>
<td>7.14 singlePalletUnload</td>
<td>182</td>
</tr>
<tr>
<td>7.15 palletStacking</td>
<td>184</td>
</tr>
<tr>
<td>7.16 palletStoring</td>
<td>186</td>
</tr>
<tr>
<td>7.17 transferToStack (part 1 of 2)</td>
<td>188</td>
</tr>
<tr>
<td>7.18 transferToStack (part 2 of 2)</td>
<td>189</td>
</tr>
<tr>
<td>7.19 moveToStack</td>
<td>191</td>
</tr>
<tr>
<td>7.20 palletOutlet (1 of 3)</td>
<td>193</td>
</tr>
<tr>
<td>7.21 palletOutlet (2 of 3)</td>
<td>194</td>
</tr>
<tr>
<td>7.22 palletOutlet (3 of 3)</td>
<td>195</td>
</tr>
<tr>
<td>7.23 palletUnstacking</td>
<td>197</td>
</tr>
<tr>
<td>7.24 palletRetrieving</td>
<td>198</td>
</tr>
<tr>
<td>7.25 transferToPlatform</td>
<td>200</td>
</tr>
<tr>
<td>7.26 moveToPlatform</td>
<td>201</td>
</tr>
<tr>
<td>7.27 truckLoading</td>
<td>202</td>
</tr>
<tr>
<td>7.28 multiplePalletLoad</td>
<td>204</td>
</tr>
<tr>
<td>7.29 singlePalletLoad</td>
<td>205</td>
</tr>
<tr>
<td>7.30 iptBrokerage</td>
<td>207</td>
</tr>
<tr>
<td>7.31 iptRental1</td>
<td>208</td>
</tr>
<tr>
<td>7.32 iptRental2</td>
<td>210</td>
</tr>
<tr>
<td>7.33 iptRental3</td>
<td>212</td>
</tr>
<tr>
<td>7.34 iptRental4</td>
<td>213</td>
</tr>
</tbody>
</table>
List of Figures

9.1 Example Logistic Contract Space ................................. 238
9.2 2-party interaction .................................................. 242
9.3 Using the Jini lookup service to locate party registrars .......... 243
9.4 Returning from a task ............................................... 245
9.5 2-stage return ......................................................... 246
9.6 A structured journal tree ............................................ 247
9.7 A flat journal tree .................................................... 248
9.8 Protocol fragment for figures 9.9 and 9.10 ....................... 254
9.9 Journal tree at $t = 1$ ................................................ 255
9.10 Journal tree at $t = 2$ .............................................. 256

A.1 Using the Jini lookup service ....................................... 287
A.2 A typical Jini service ............................................... 288
A.3 An alternative Jini service .......................................... 290
List of Figures
Part I

On-line logistics: WHY and HOWTO$^1$
This thesis describes recent work we have done in the field of on-line logistics: the design, use, and implementation of the LINC logistic contracting language. The thesis is divided into five parts, the first part being an overall description of the field and of our own contribution to it. The remaining parts provide a detailed description of the LINC language, two complete case studies, an introduction to our implementation of the language, and an evaluation, respectively.

In this part, chapter 1 sets off with a description of the current situation in logistics. After describing the shortcomings that are inherent to the current approach, it argues that a change of paradigm is inevitable. A new paradigm, called *New Logistics* by Evers [ELL99, Eve00], embraces the network (e.g., Internet) as the appropriate means for organizing virtual logistic enterprises. The proposed paradigm will then be compared to related developments in computing, e.g., web services [Kay03]. Next, chapter 2 will introduce the AgileFrames framework, which formed the context for our own research. The framework, originally designed by Evers [ELL99, Eve00], is based upon the New Logistics paradigm. Whereas the lower levels of AgileFrames (i.e., TrAcEs and FoRcEs) have already been implemented by Lindeijer [Lin03], the top layer (logistic ‘control’) has been the topic of our own research. Chapter 3 presents a new design method for logistic control: *Actor Interaction Modeling (AIM)*. The method is based upon the specification of the interactions between logistic actors; these specifications take the form of logistic service contracts. Chapter 3 ends with an outline of an organizing principle for service contracts, called the *Logistic Contract Space*. Finally, chapter 4 describes the main subject of our research: the design of a language for writing logistic service contracts. Following a short introduction to the research goals as they are determined by the new paradigm and by AgileFrames in particular, it presents a set of requirements for the design of the LINC language.

---

1The term “HOWTO” was borrowed from the Linux world, where it refers to a document describing a particular hardware or software issue. Most HOWTOs are published as part of the *The Linux Documentation Project (TLDP)* [TLD].
Chapter 1

Introduction

In this chapter we will provide a sketch of recent developments within the field of logistics and focus on the change of paradigm that is both enabled and required by recent technological, economical, and societal developments. We will describe the developments that have lead to our current undertaking: the design of a language and related tools for logistic contracting. Our description will clearly demonstrate that the approach comes naturally and that the time was right for it: similar developments took place concurrently, albeit independently. The common goal of these developments is to find a new approach to the growing need for integration of software systems, both inside a single company and across a number of companies that together form a supply-chain.

First, section 1.1 will introduce the challenge that must be faced by logistics, i.e., to meet the requirements that are posed by a changing society. Next, section 1.2 will introduce Enterprise Resource Planning, the paradigm that still dominates logistic systems design, and point out its shortcomings in relation to recent developments. Following that, section 1.3 will lay out the requirements for the paradigm that is to replace it, and section 1.4 will describe the proposed new paradigm, that we call New Logistics. Section 1.6 we will mention a number of similar developments that took place more or less concurrently with our own research. This section will also mention the most important differences between our approach and the related developments. Section 1.5 will outline the goals of this research. Finally, section 1.7 will end this chapter by presenting the organization of this thesis.

1.1 The challenge of on-line logistics

Logistics is defined in this thesis as the providing of services delivered by service providers (or shortly: providers) to clients on their demand, while using resources (see also [ELL00]). This definition can be applied to the various subfields of logistics, such as chain logistics (supply, transportation, storage, and distribution of commodities), system logistics (manufacturing and maintenance of technical systems and the supply of related goods and materials), and service logistics (non-material activities such as mass transportation, health care, security services, financial services, etc.). This thesis addresses some issues raised by
on-line logistics, a field that is growing rapidly and that is bound to replace conventional (network-based or otherwise) logistic systems within the next 5-8 years or so. As we will explain in more detail, the field of logistics is facing a major change of paradigm: no longer is the network that connects computers a tool to ease certain operations; instead, it has become central to how logistic operations are conducted, enabling an approach and resulting in benefits that are unattainable using more conventional techniques. While this view itself is not so new (Sun Microsystems has already been using the slogan ‘The network is the computer’ for more than a decade now), technology is finally coming to a point that allows the theory to be implemented.

1.2 Enterprise Resource Planning

Logistic systems design is still dominated by Enterprise Resource Planning [BC96, Loz98]: the logistics system is interpreted more or less as a regular enterprise and the usual techniques, focused on the integration of the software systems supporting the organization, are used to to design the supporting software. The primary focus of ERP is on the business processes themselves, and even though it is recognized that an enterprise consists of a (usually large) number of relatively stand-alone sub-processes to which certain activities have been delegated, the individual actors that together make up the enterprise are taken for granted: they form a fixed and irreplaceable part of that enterprise and are organized in a frozen and unchangeable hierarchy. The task of these processes is to ‘instruct’, rather than to ‘inform’ [Eve00]. Usually, such an enterprise has been founded to solve a very specific problem, i.e., to produce a specific range of products and/or to provide a specific kind of services: it simply does not have the flexibility to do anything else. As a consequence, the design and maintenance of software for such enterprises tends to be centralized too, even when the software will run in a distributed fashion. Most current logistic systems, if not single monolithic systems, comply to this description and consist of a number of individual applications that, while running at different locations, still behave in a closely-coupled fashion. Such enterprise systems are nicely characterized by the following quote from the foreword of [CDK93]:

"Organizations are moving to exploit computing systems more effectively, and this is giving rise to a new class of large-scale distributed applications. These enterprise computing systems offer highly integrated, highly reliable computing to users who may be physically separated by large distances and interacting with a multiplicity of computing devices. They combine large numbers of independently executing programs into an (apparently) seamless whole, and often provide services critical to the organization. The development of software for such systems poses difficult challenges, particularly because of the need to respond dynamically to failures and recoveries. This is further complicated by the constraint that the system behave consistently regardless of where it is accessed."
1.3. The need for a new paradigm

Enterprise systems ... are characterized by a close, direct coupling of application programs running on multiple platforms in a networked environment. The programs involved will generally coordinate their actions to present a consistent view of the system to users. Indeed, it is common for such a system to mimic a single highly reliable program, despite the decentralized nature of the underlying software."

The above description does not only represent the academic views, it is also an accurate description of daily IT practice (which has a dynamics of its own, sometimes quite distinct from what is considered relevant in the academic world). Consider for example the following definition of ERP, which was taken from the ERP-fans web-site [ERP] (the underlinings were already present in the original definition):

“Software solution that addresses the enterprise needs taking the process view of an organization to meet the organizational goals tightly integrating all functions of an enterprise.”

It is precisely the need for tight integration that forms the weak spot of the ERP approach. By being based upon a one-time analysis of what the client demands are supposed to be instead of being directly driven by these demands, it is very difficult to adapt an enterprise to reflect any changes in these demands. And to scale or restructure such an enterprise in response to changes in demand or requirements is more difficult than just ‘hiring’ or ‘firing’ a few actors. In an article called ‘Elephants Rarely Pirouette’¹, Brace and Rzevsky [BR98] state that the key issues in logistic systems design are responsiveness, adaptation, and replacement of central control by coordination between activities that are being outsourced, whereas ERP only supports vertical integration. According to Evers [Eve00], ERP systems, by their very nature, do not support flexible adaptation and improvisation, and they do not support an open society of actors that can reorganize itself in response to changing client demands. More recently, in a book called ‘Loosely coupled’, Kaye [Kaye03] comes to similar conclusions. In conclusion, ERP and similar centralized and/or tightly integrated systems are:

- difficult to adapt to changes and restructurings,
- less robust against disturbances,
- difficult to scale.

1.3 The need for a new paradigm

Let us start by saying that running an enterprise is no longer thinkable without the support of (something like) ERP. However, the main problem with applying the ERP approach to

¹ A ‘Google’ search for this article however turned up some evidence that elephants actually do pirouette, a fact that is strongly opposed to by animal rights activists.
logistics is that a logistic system is not necessarily a regular enterprise, and that software that is designed in a way that tries to achieve tight integration cannot cope with the changes that are bound to occur in a real logistic environment. Not only do we want the objects that are handled or produced by the logistic process to become active components in the running systems, we also need to handle logistic actors that move around, cease to exist, or cannot keep the promises they made. In all these cases, we want to make sure that the objects involved (be them the results of a manufacturing process or the object to be transported) are delivered with a minimum of delay; to accomplish this, we may have to switch to other parties or to invoke the help of additional parties. Other changes that must be anticipated are changes initiated by the client: change of amount or volume, total cancellation of the order, qualitative changes in the product or service, and changes regarding the delivery time frame. For logistic systems, such changes are however "business as usual". Systems that cannot cope with such disturbances are doomed to fail.

A similar move from process oriented control to client demand control is happening in industry, where markets become dominated by fast and accurate delivery of product assemblies that are specified by the client [IG98]. Additional arguments come from Olin et al. [OGK99], who have observed that (at least in automotive industry) different manufacturers are merging into portfolio based companies that shift their focus from in-house R&D and manufacturing depth to assembly and marketing. They argue that the resulting enterprises are not well served by hierarchical planning tools and observe that global network-based information exchange is a potent competitive instrument enabling concurrent product development at a global scale. Their conclusion is that the shift from vertically integrated companies to modular supplier consortia poses new challenges that are best coped with by increasing the individual supplier's autonomy while encouraging the formation of 'virtual' enterprises. Their arguments are fully compatible with those of Brace and Rzevsky [BR98], who state it has been clear for quite some time now that productive systems in which intelligence is distributed rather than centralized are more responsive and more economic of resources. In their words: "economy of scale has been replaced by economy of scope and in turn by economy of governance". Finally, according to [BMB99], the openness of the global network results in logistics moving towards a market 'anarchy', where deals have to be done instantaneously. For every possible deal, there is a window of opportunity in which the deal can be realized [IG98]. The time frame in which an opportunity can be caught is equal to this window of opportunity minus the time that is needed to configure the (supply) chain and assign the assets. The latter time should therefore be as short as possible: yet another reason why the flexibility requirement has become so all-important. According to Evers [Eve04], just extending ERP with communication facilities is not good enough, for in order to support the continuous restructurings that are the result of globalization and of increased client orientation, the components themselves have to be designed differently\(^2\): instead of assigning all coordination tasks to the humans who design and/or configure the (logistic) systems, these tasks should be deferred to the execution stage of such systems by equipping them with the means to select the parties they need to cooperate with and to

\(^2\)This viewpoint only started to emerge in the mid nineties, during the course and shortly after Evers' participation in the CMSO project [Bee92].
coordinate the actions that are required to achieve the intended goals [Sin01, page VI]). In conclusion, new logistic systems must be managed as if they are operating in a continuous state of improvisation, and the business processes themselves can no longer be taken as a stable basis for their design [Eve00].

While most of the above arguments were presented in the context of manufacturing processes, their relevance extends to all logistic systems mentioned in the start of section 1.1. In particular, they are relevant to service delivery in general, where increasing globalization has reached a level that asks for a new approach. The following section will present this approach, which we call New Logistics, in more detail.

1.4 New Logistics

Recent technological and economical developments hold the promise of a qualitative change, as they allow the emphasis to be moved from using the available means to speed up existing mechanisms to using them in ways that were not feasible before. Data being exchanged is no longer passive but can be used to change the behaviour of the logistic applications themselves and/or to restructure the client/provider network itself. The growth of fast communication systems (e.g., the Internet) and the recent advances with respect to service discovery (as offered by CORBA [OMG02a], Jini [AOS+99], P2P [OTG02], etc.) allow for self-organizing, self-repairing network organizations. These new technologies provide the means for a nearly transparent globalization of logistics, and are therefore ideal to implement systems that have the flexibility to support cellular manufacturing, virtual enterprises, and ultimately: client control. They form the crucial enabling factor for a new logistic paradigm (which was dubbed New Logistics by Evers [ELL99, Eve00]) in which logistic operations are carried out by societies of autonomous actors (defined by him as software modules that can be activated at any time, from any place, at any place [Eve00]), that interact with each other in an open environment. According to this new paradigm, logistics will become communicative, agile, and real-time, and logistic systems will become client oriented rather than process oriented [ELL99, Eve00].

Dividing an enterprise into autonomous actors that cooperate to solve independent problems in small groups is fundamental to the architecture of new logistic systems, and necessary to overcome the inherent complexity problems that are associated with scaling up a centrally controlled enterprise (virtual or not) [Eve00]. At the the same time, such actor groups allow for flexible regrouping in response to changing demands and requirements. Of course this requires that the actors (and the processes and organizations they represent) are not only replaceable but also fully and transparently interchangeable. Together, these actors will form virtual logistic enterprises. In order to achieve the desired flexibility, it is mandatory to allow the client to initiate and direct the transactions that happen in a virtual enterprise. However, in order to achieve the necessary operational flexibility, the system should use client service requests to anticipate, not to fixate: decisions on transactions have to be postponed for as long as possible given the commitment and the efficiency requirements of
the service providing process [Eve00].

Logistic systems that are designed according to the above principles and consist of flexible groups of cooperating actors will (at least in principle) be:

- **changeable**, i.e., reactive to changes in client demands, to modifications of running orders (including cancellation), to new business opportunities, new infrastructure, and to (dis)appearance of services that can be subcontracted, even when the system is fully operational,

- **scalable**, both up and down\(^3\), to any size without substantial size-related loss of efficiency and effectiveness,

- **responsive**, meaning that the system can be (re)structured quickly to disturbances and changes like those mentioned above,

- **robust**, i.e., able to cope with unexpected situations by allowing actors to come to help to existing actors when necessary, and by being able to fall back to human emergency handling (through an appropriate user interface, of course),

- **network oriented**, i.e., able to arrange logistic transactions over the medium that has become pervasive: the Internet, and transferring the flexibility of modern distributed software systems to the logistics applications themselves, thereby allowing them to adapt to rapidly changing situations,

- operating in **real time** in order to control a logistic operation as it unfolds,

- moving the focus from **problem solving** to **opportunity catching** and use ‘ad-hoc standardization’ to enhance compatibility between providers and clients so they can act swiftly in response to an opportunity when it occurs.

Together, these properties allow for a reversal of the traditional way of organizing things: instead of first building a (physical or virtual) enterprise and then waiting in the hope there might appear some work to be done, individual small-scale clients and providers may present themselves to ‘cyberspace’ and negotiate to form *ad-hoc* enterprises in response to a specific business opportunity. In this view a logistics application is like an *enterprise system*\(^*\) where the enterprise in question is much more volatile than a conventional enterprise: it is established for the job at hand, and is a temporary constellation of logistic actors that use a common software platform to locate each other, to negotiate about the services they need c.q. offer, and to establish a temporary working relationship between each other.

\(^3\)Usually, scalability is taken to mean that a system can *scale up* in order to profit from the ‘advantages of scale’. This is not enough, though, since we want to be prepared for client demands that go down and orders that get canceled, in which case we want to *scale down* and allocate the idle capacity to another task as quickly as possible.
1.5 Research outline

Obviously the new approach has to be supported by a software platform to make it easy for a party to join or leave a virtual enterprise and to provide or use services to or from other parties in a reliable and trustful way. The absence of a permanent organizational structure requires the working relationship of a virtual enterprise to be formalized in new ways; according to Brace [BR98], cellular manufacturing “needs a combination of a common language, a shared highway, and a set of common laws. Cells have to agree on partial transparency of their essential data to other cells, and have to agree on common protocols”.

In [Ell99, Eve00], Evers has proposed a generic architecture supported by a generic tool set to create logistic control systems that are both communicative and flexible. This platform, called AgileFrames, is designed to support logistics operations from the lowest level of machine control to the highest level of logistic contracting (see section 2.1 for details). The top layer of this architecture is realized as a logistic contracting layer, i.e., the “common language, shared highway, and common laws” of AgileFrames. Logistic contracts can be developed by any party (provider or client) and implemented by whatever party that wants to provide or use the service. They allow the focus to shift from the core business processes themselves to the matching of supplies to demands, and result in systems that are more flexible, more reliable, and easier to develop and maintain. Finally, they replace conventional standardization efforts, which would be too slow and too static to cope with fast and frequent changes; in effect, they achieve what could well be called ‘ad-hoc standardization’.

The main purpose of logistic contracts is to define the skills* of the parties involved in a particular service as well as a detailed protocol describing the interactions between these parties. In the context of this study, all parties are taken to correspond to software objects; since logistic operations are usually non-local, such objects may reside anywhere or even be mobile. The usual way to define an object’s capabilities is to define its interface or signature*. Such an interface provides a static description of the object. Restrictions on an object’s dynamic behaviour may or may not be present; such restrictions would take the form of preconditions, postconditions, and invariants. Hence, an object’s dynamic behaviour is captured in a set of static constraints. While this approach may be perfectly satisfactory in the context of pure software design, it is less so when designing logistic systems. Logistic systems are inherently distributed, and the various actor objects will be maintained by different parties; it would be totally unacceptable if a logistic operation would just terminate (throwing an exception) after some error was detected in one of the actors involved in the operation. Moreover, a purely logical approach to defining actor behaviour does not match well with the way logistic operations work: timing is all-important to such operations, and restrictions on order, concurrency, and timing form a natural part of logistic designs. Our approach therefore was to design a contracting language that integrates the logical aspect with the temporal one, and that does so in a way that reflects the reasoning of the logistic engineer: a contract defines an actor’s skills (i.e., the capabilities that a requesting actor may ‘order’ from the providing actor) and that furthermore states when it is proper to request each skill. Indeed, one might envisage the protocol as a play, and
an actor as a stage actor: the script determines when it is proper for an actor to shout something towards the other actors (or to the public, for that matter). Yet another way to look at the protocol is to interpret it as a dialogue-grammar, and the actual sequence of interactions as a dialogue: just like a conventional grammar, the protocol would draw a distinction between correct interactions and incorrect ones. Since logistic contracts as described here define all relevant aspects of a logistic operations, they become reusable entities themselves: it becomes easy to envisage a pool of common contracts that may be implemented by companies that want to advertise themselves as a provider or client (!) of the described service.

When approaching a research subject like the one described in this thesis, many choices must be made: the topic is rich enough to fill not just one thesis, but several other books and articles as well. Their titles would include, e.g.: 'Formal semantics of the LINC language', 'Ambiguity in protocol languages', 'Grammatical formalisms and Coordination', 'The implementation of protocol languages', 'Discovery mechanisms for logistic actors', 'Transaction mechanisms and service protocols', 'Security of logistic contracting systems', and 'Legal aspects of logistic contracting systems'. Of course, this list could be extended at will. The selection of topics that is included in the current thesis is inspired by our strong conviction that at this point it is most useful to have a language to describe the interactions between loosely coupled (logistic) actors and to gain experience using it to design real systems; doing so will undoubtedly change some of the views that are now held. The design of such a contracting language has therefore been our main research issue.

The proposed language is validated by applying it to perform a number of case studies and by implementing a prototype for the validation of interactions on the basis of compiled service contracts. While the logistic design presented as the first case study is only intended to introduce the reader to the design method, the design presented as the second case study is meant to be representative; even though some details may need to be elaborated further, the given contracts may be taken as a starting point for the implementation of a real system. A secondary research goal was to design a compiler for the language and a validation system for the interactions taking place during execution. The main purpose of the prototype was not to obtain an efficient production system though, but to demonstrate that the language can indeed be implemented. In a number of cases that are indicated in the text, the prototype implementation has inspired a change of the language.

1.6 Related developments

While the reasons behind the need for the new paradigm have been described in detail in the previous sections, here we observe that a similar change is underway in a number of related fields, both at the application level and at the research level. Applications can be found in the areas of both manufacturing [JAS99], e-business (C2B) and business-to-business (B2B) systems [DP98, SDN+00, DDK+01]. These application areas are coming together in what is now referred to as the Web Services approach: in [Kay03], it is argued that organizations
have to let their (enterprise) systems evolve to 'web services' in order for them to stay competitive and meet the demands of their customers. In fact, the same development is at the basis of our own research: during the early nineties, when TU Delft joined Actis in Stuttgart GMH, Lucas Automotive Ltd., Alcatel, and a selection of other companies and research institutes in the CMSO project [Eea92], it was already recognized that enterprise systems should be designed to reflect a changing society: while industries had just started to move from 'stock' production to demand driven production, the latter would only suffice for some time and be superseded by systems that allow customers to order highly personalized products by selecting from a vast number of (standardized) options [Eea92, BR98, JAS99]. By the end of the CMSO project, it was clear that software systems should be adapted to meet the changing requirements. Like in the other fields we mentioned, the solution was expected to come from a separation of concerns, dividing applications into process logic and interaction logic. It is the explicit specification of the latter interaction logic that allows subsystems ('services') to become loosely coupled. Loosely coupled systems can be (re)arranged quickly in response to specific client demands and avoid the need for tight integration of all systems that make up a supply chain, a task that is clearly infeasible in the light of the changeable requirements that are becoming common practice (see also sections 1.2 and 1.3).

At the research level, the field of coordination languages has gained importance as part of the theory of distributed computing (see e.g. [FF84, GC92, Arb98, PA98, OZKT01]). Whereas in traditional systems cooperation between concurrent processes was modeled by the communication primitives that happened to be supported by the operating system used, coordination language research approaches the problem more systematically by replacing such ad-hoc primitives by primitives that are part of a well-defined cooperation/coordination model. Coordination languages provide a notation for specifying and a means for managing the interactions between computing agents [BCGZ01]. Coordination language research focuses on the definition of suitable primitives for languages describing the interactions between distributed objects and on a formal semantics for such languages. Since in modern systems the passive role of the user as an observer is being replaced by a role in which he is an active component of the system, the traditional barrier between users and applications is being removed; the authors of [GC92] use the term ensembles to refer to groups of cooperating actors and specifically include both animate and artificial actors. As the behaviour of the actors of such open systems might be unpredictable (they might be humans after all), such systems can not be modeled by Turing Machines. The concept of an Interaction Machine (i.e., a Turing Machine extended with interactive I/O) was introduced to describe such an open system [Weg95, Weg98]: the resulting systems are essentially under-specified [Arb98]. The objectives of coordination language research are very similar indeed to those of commercial applications: to devise a way to separate the internals of concurrent objects from their interactions, thereby allowing them to set up (temporary) task forces to achieve a specific goal. The separation between interaction logic and process logic also happens to separate concurrency aspects from the much simpler.

The first reference shows that coordination language research originated quite early; in that reference, it was referred to as 'Coordinated Computing'.
single-threaded execution that takes place within the objects that make up a concurrent application; it thereby contributes to the design of such applications. Whereas the latter aspect is not always explicitly mentioned in the more practice-oriented literature, it is no less significant in that area.

In between the two extremes of applications and research, and combining aspects of both, one can find the so-called scripting languages [NTdMS91]; scripting languages are often mentioned in combination with work-flow systems (see e.g. [MPN97, MPN98]).

Differences with related approaches

As readers will notice, there is some divergence between our proposal and some of the related developments that took place concurrently, most notably Web Services [Kay03]. Before going into the details of the differences, we want to say that we not think that the gap between the two approaches is as large as it may appear at first sight: we fully share the views of e.g., [Kay03] concerning the merits of loose coupling; it is the way that we intend to achieve this loose coupling that is different. In particular, service contracts achieve the effect of decoupling process logic from interaction logic without giving up the possibility of synchronous control. While most logistic interactions might not need such synchronous control, it might be essential for lower level services. At the very least, it allows service contracts to be used ‘all the way down’, i.e., not only in high-level logistic service contracts but also in the lower levels of traffic and machine control (e.g., TrAcEs and FoReEs, see section 2.1 and [Lin03]). Future versions of the LiNC language and implementation are expected to provide support for asynchronous control by incorporating both synchronous and asynchronous interaction (see also section 8.7.1).

Our approach also differs from e.g. TPA’s [DP98, SDN+00, DDK+01] and from web services by selecting a software platform that is language-centric (i.e., based upon a specific implementation language) instead of language independent. The main reason behind this choice is also the need for tighter control than is offered by language-independent systems: at the lower level of machine control there is a need to treat actor behaviour and operational constraints as data that can be exchanged between actors (see also [Lin03]). Thus, while being more flexible than the language-independent approach at the lower levels, our design is less flexible at the higher levels by being tied to a particular implementation language. Future versions will probably provide for XML-based communication mechanism also, thereby providing interoperability in circumstances where code downloading is not a requirement (see also section 8.7.2)

One aspect in which our approach has a definite advantage over the other published languages and platform, is the protocol specification part of logistic service contracts: the abstraction level of a grammar-based language is higher than the state transition based approach embodied by many other approaches (e.g. tpaML [DDK+01], Manifold [Arb98], COOL [Che98]). This difference has a positive effect on both the reading and writing of complex interaction patterns, and has direct consequences for the correctness and main-
tainability of the specifications themselves as well as that of the complying actors.

1.7 Organization of this thesis

This thesis is divided into five parts. In part I we set out to describe the field which we call 'New Logistics', i.e., the direction into which (e.g. distribution) logistics is developing now that networking has become commonplace and the world has become highly interconnected. This part will end by describing our own goals in this field, i.e., the design of a logistic contracting language. At the end of this part we will present a set of requirements that the design has to fulfill, and define a set of criteria that will be used to evaluate the design.

Part II will present the complete language that was designed in accordance to the requirements given at the end of part I. It is a language reference as well as a design rationale: many descriptions of language constructs are accompanied by a short section detailing the relevant decisions that were made. Whenever appropriate, these 'rationale' sections will show how the design has been influenced by our attempts to use or implement the language construct.

Part III will then continue by using the language to conduct two different case studies. Whereas the first case study is relatively small-scale and not designed to accurately reflect a real-world system, the second case study is a comprehensive example of the service contracts that define a real implementable system. The goal of these case studies is to evaluate the LinC language and to accredit the use of logistic contracting as a design method. Following a short presentation of the global design, each case study will present all service contracts, most of them annotated with extensive comments detailing the use of the LinC languages. If a certain design problem has influenced the language, this fact will be noted in the comment.

Part IV will look at our language and system from the perspective of distributed programming systems. Building upon a basic description of the evolution of distributed systems and of the available software technology, it will show how the LinC platform can be implemented. The goals of this chapter is to show that the platform can indeed be implemented, and as a side-effect, to provide a starting point for implementors.

Finally, part V will evaluate the design according to the criteria that were set out at the end of part I and present the most important observations that were made on the basis of the case studies in part III.

This thesis concludes with 2 appendices: the first one is a glossary of terms from logistics, computing science, or this thesis itself. The first occurrence of a term that is described in the glossary can be recognized in the text from the trailing *asterisk* and the use of the *italic* font. The second appendix provides an introduction to Java RMI and Jini, which together provide a basis for the LinC implementation.
Chapter 1. Introduction
Chapter 2

Context of our research

The research described in this thesis was focused on one specific way to support the development of New Logistic systems as described in section 1.4, viz., the use of logistic service contracts to specify services to be offered by logistic actors. The research took place in the context of AgileFrames, a software design method and associated development environment for logistic systems that has been under development at the department of Transport Engineering and Logistics of the Delft University since the late nineties [ELL99, Eve00]. AgileFrames consists of three distinct layers, of which two layers (i.e., TrAcEs and FoRcEs [ELL99, Eve00, Lin03]) were already more or less completed when our own research took off. Section 2.1 will present an overview of AgileFrames. Following that, the different layers of AgileFrames will be described in more detail: section 2.2 will introduce FoRcEs; next, section 2.3 will introduce TrAcEs; and finally, section 2.4 will introduce LiNC. The latter layer, which forms the top level of AgileFrames, will be the subject of the rest of this thesis.

2.1 AgileFrames

This section will provide a short introduction to AgileFrames, which formed the context of our own research. AgileFrames [ELL99, Eve00, Lin03] is designed as a platform for developing New Logistics applications. The platform consists of a design method and an associated development environment, and offers a generic architecture for building software to control automated logistic systems. The AgileFrames architecture is service oriented, which means that the services to be provided by the intended logistic system are taken as the starting point for the design process. In order to meet the requirements that follow from the New Logistics paradigm described in section 1.4, the following aspects have played an important role in the design of AgileFrames [Eve00]: openness, scalability, concurrency, transparency, heterogeneity, security, and failure handling.

AgileFrames distinguishes between a real logistic world consisting of clients, providers, services, service objects (the entities affected by the service), service monitors (parties interested in the unfolding service), and infrastructure, and a virtual logistic world (consisting of software models representing those clients, providers, services, etc.). Logistic control is
exercised in the virtual world; the physical world is expected to follow (see below). AgileFrames defines three distinct levels of control, where each level corresponds to a specific (set of) competence(s) and a specific time scale:

- at the top level, *logistic control*\(^1\) is responsible for the cooperation between logistic actors; this is the level that defines the aforementioned services, contracts, and actors,

- at the middle level, *traffic control* is responsible for synchronized access of shared infrastructure by traffic actors; such infrastructure may consist of manufacturing capacity, storage capacity, transportation capacity, maneuvering space, etc.,

- at the bottom level, *machine control* is responsible for the setting of physical actuators and the reading of sensory feedback data in order to implement the desired machine behaviour and keep the correspondence between the virtual world and the physical world within well-defined limits.

The advantage of distinguishing these layers is a clear allocation of the various tasks to dedicated actors and hence a clear separation of responsibilities. Each level of AgileFrames corresponds to a distinct software layer: machine control is handled by FoRcEs [Lin03], traffic control is handled by TrAcEs [Lin03], and logistic control is handled by LiNC, which is the topic of this thesis. The mutual relations between the three layers as well as the interfaces between the layers and their environment are depicted in figure 2.1, which clearly shows that each level is controlling a distinct aspect of the physical world: goods or services

![Diagram of AgileFrames](image)

**Figure 2.1: AgileFrames**

at the top level, infrastructure at the middle level, and machines at the bottom level; the conflict-handling middle layer may be skipped if there is no infrastructure to share. Furthermore, each level is connected to a set of monitors that provide feedback information and control signals from the outside world. Control in each layer is autonomous (to a certain degree), adaptive to external control and events, and it anticipates future events on the basis of its internal model of the world.

\(^1\)Note that we do not use 'control' in the traditional sense: there is no central authority that controls logistic actors. Instead we take logistic contracts as the supervising entities that guard the cooperation of one or more parties. These parties have their own independent control; however, they accept the rules that are posed by the contract.
Each level corresponds to a separate development environment which includes a set of software components supporting the core notions of the AgileFrames design method. These components have been implemented in the Java programming language, and make extensive use of remote polymorphism. The interfaces between the various layers are as thin as possible, and are mostly event-based. Apart from the general goals of shortening the logistic development cycle and increasing responsiveness, speed, reliability, and safety, each software level also has to meet the following specific requirements (see also [Eve00]):

- control must be agile to adapt to changing requirements, changing client demands, (lack of) logistic actor performance, even while the system is in operation.

- control must be robust to cope with exceptional circumstances like hardware malfunction, extreme weather conditions, traffic congestion, client misbehaviour, subcontractor failure, etc.,

- control must be real time, meaning that it must be able to keep up with the unfolding operation\(^2\),

- control must be decentralized in order to avoid the complexity explosion that is usually the result of scaling up,

- control must be integrated with simulation in order to support the development and testing of new logistic systems.

The different layers will be described in more detail below.

### 2.2 FoRcEs

FoRcEs (Function Oriented Robot Control and Engineering System) is a design method for the definition of fine grained control of arbitrary machines [Eve00, ELW01, Lin03]. The method, which is supplemented by a software development toolkit, is primarily intended for use by logistic engineers to create functional specifications of what the controlled machines should do; it hides the most difficult aspects of machine programming (such as threading) from the logistic designer. A FoRcEs specification takes the form of a progression through a sequence of states defining the relation between the machine and its environment. Transitions between successive states are specified in the form of executable moves, where executing a move consists of following a prescribed trace over time. This trace specifies the route\(^3\) to be followed by the machine. The move uses event based interaction with its environment to guide the machine through the trace: flags are used to pass state information about the move, such as the passing of a certain point in space or time; precautions

\(^2\)The exact meaning of real time is therefore different at each level.

\(^3\)The term ‘route’ is used here in a general sense to denote the evolving relation between a machine and its environment.
are used internally to a move to adapt the speed to external circumstances. Together, these mechanisms illustrate how a discrete event oriented view of progress can be related to a continuous function oriented model.

A FoRcEs specification consists of a set of scripts that are interpreted by a controlling process (the so-called Machine Function Driver), whose outputs drive the physical actuators, motors, etc; the inputs of this process consist of feedback devices monitoring physical processes as well as software events coming from external sources that monitor the machine’s behaviour and state. Systems designed using the FoRcEs modeling method and toolkit can drive real (i.e., physical) devices, simulated devices, or any combination of these.

2.3 TrAcEs

TrAcEs (Traffic Control and Engineering System) is a design method for the definition of agile, scalable, flexible, and safe traffic control systems for use in well structured environments [LE99, Lin03]. The method, which is again supplemented by a software development toolkit, can be used to coordinate concurrent use of shared infrastructure by a number of machines (e.g., vehicles) controlled by e.g. FoRcEs, including routing\(^4\). The TrAcEs software toolkit therefore has to offer a means to handle conflicting demands for exclusive use of infrastructure elements.

TrAcEs describes the physical infrastructure in the form of scenes, which are software representations of the physical world, and by describing a machine’s route as a script within a scene. A scene is a hierarchical structure consisting of infrastructure components, where each such component is associated with a traffic control agent (see [Lin03]). Access to shared infrastructure elements is protected by semaphores: the agent has to request access, after which the TrAcEs semaphore (an extension of the common semaphore concept that is pervasive in concurrent software) grants access if the situation and the traffic rules allow. Until then, the requester will have to wait (the required deceleration or acceleration actions belong to the domain of FoRcEs). After all potential conflicts on the use of shared resources have been resolved, a route is generated and passed to a machine in the form of a sequence of moves (see above). The moves are then executed autonomously by the machine that is to follow the route.

2.4 LinC

The top level of the AgileFrames control hierarchy is concerned with the definition of logistic services and actors. Services may be provided at many levels, ranging from customer-level services (e.g., transportation of a payload) through intermediate level services like reservation of infrastructure (e.g., renting container space from a freight carrier), to machine level

\(^4\)Machines may or may not communicate to each other to perform low-level control such as distance regulation between two or more individual vehicles.
services (such as pallet unloading using an automated pallet truck). All these services are ultimately implemented by means of 'machines' that share common infrastructure elements or capacity (e.g., physical space, weight carrying capacity, moving capacity, production capacity, etc.); in carrying out their tasks, the logistic actors will therefore rely on the lower levels of AgileFrames.

As was explained in section 1.5, the goal of our research was to develop a design method as well as a software development toolkit for the specification and implementation of logistic services within the AgileFrames framework. The design method will be used to specify virtual logistic enterprises by defining the skills of the logistic actors involved and the interactions that may take place between them. Systems designed using the method and tool kit need to combine the advantages of enterprise systems (i.e., offering the user a complete and consistent view of the system, providing a very high level of availability and reliability) with the seemingly conflicting needs of a very agile environment (e.g., providing ways to locate other parties and form temporary networks with them, providing ways to restructure a running operation). Chapter 3 will introduce the design method (called Actor Interaction Modeling) that we propose and define the underlying concepts: logistic actors, logistic service contracts, and the Logistic Contract Space. The requirements for a language that can be used to write service contracts will be presented in chapter 4. The language itself will be the subject of part II, and a sketch of a prototype implementation will be presented in part IV.
Chapter 2. Context of our research
Chapter 3

Logistic contracting

In this chapter we will outline the basic concepts underlying logistic contracting, building upon the general ideas about New Logistics presented in section 1.4. We will argue that in order to design the required flexibility into a logistic system, we have to start the logistic design by defining the interactions between the parties that will make up the system. Section 3.1 introduce a new design method, called Actor Interaction Modeling, that is based upon such a specification of the interactions. Subsequent sections will detail the concepts that form the basis of Actor Interaction Modeling: section 3.2 will introduce logistic actors, while section 3.3 will explain logistic contracts in detail. In order to be (re)usable, logistic contracts (and sometimes the code corresponding to actors as well) must reside in a well-structured 'contract space' that is accessible from anywhere. Section 3.4 provides a short description of such a contract space, the Logistic Contract Space.

3.1 Actor Interaction Modeling (AIM)$^1$

As we have argued in section 1.4, the new logistic paradigm is based upon logistic parties forming an 'ad-hoc' enterprise on the basis of a specific demand. Furthermore, such a virtual enterprise must be established in record time, as the opportunity will go to other actors if their response is faster. Finally, the actors have to be able to restructure the virtual enterprise if circumstances demand; actors may be asked to join or leave the enterprise, or to provide new services or extend existing services. The physical location of an actor may or may not play a role in the enterprise; most of the time, their capacity will be more important to the goals of the virtual enterprise. As we saw in sections 1.4 and 1.6, the actors have to be loosely coupled in order achieve the required flexibility; at the same time, their behaviour should conform to mutually agreed standards in order to avoid anarchy. In AIM, these mutually agreed standards take the form of logistic service contracts. Using the words of [BR98], we say that service contracts play the role of the 'common laws' (see also

---

$^1$The words actor and agent are both used to denote software entities that are able to perform some useful task and that cooperate and communicate with other actors/agents to achieve a common objective. Although the terms are often used interchangeably, we prefer the term 'actor' over 'agent' because the latter word carries some additional (AI-related) connotations that we would like to avoid in the current context.
section 1.5); the language to write contracts in then is the ‘common language’ (see [BR98] and 1.5), while the implementation platform will provide the ‘shared highway’ [BR98]. The design of such a language and an implementation have been the subjects of our research; both issues will be addressed in the remainder of this thesis.

3.1.1 The role of service contracts

In sections 1.4, 1.5 and 2.4 it was already mentioned that our approach will be based upon logistic services, to be used or provided by logistic actors (i.e., software entities representing companies, machines, etc.). Since location is not an issue and the network is to play an important role, any party can in principle become an actor in a New Logistics operation. The problem with distributed operations is, however, that the reliability of the operation as a whole might be severely threatened by actors that are not fully reliable themselves, that disappear without notice, or that just behave unexpectedly. None of these threats is far-fetched, as any regular Internet user will readily confirm. Given the number of parties involved in a typical logistic operation and the volumes of goods and money going around, it is absolutely necessary to devise a mechanism that safeguards the enterprise, that monitors an ongoing operation, and that provides a basis for legal or financial action if things go wrong despite all this. If possible, the same mechanism must be used to help establishing the virtual enterprise and to restructure it if necessary.

The hypothesis embodied by AgileFrames is that logistic contracts provide such a mechanism, and that they form the key to realizing logistic operations in a way that is compatible with the requirements of New Logistics. In AgileFrames, a logistic design takes the form of a (set of) service contracts that are published for anyone to see, and that can be adopted by potential service providers, clients, or any other parties implementing a role defined by the contract. In principle, any party can create such contracts and ‘donate’ them to the logistic society; alternatively, a so-called user community process can design, discuss, and standardize new contracts. In both cases, the service contracts will become part of an open ‘library’, available to anybody. Interested parties can develop applications supporting a particular role in the contract; if they want, they can use the applications themselves by offering to play that role in a logistic operation. Alternatively, they can donate their application(s) to the library as free-ware, or sell them as commercial software products. Other parties can then obtain such existing code and offer to play the corresponding role.

3.1.2 The role of logistic actors

In the above setup, parties register themselves announcing the role(s) they want to play in one or more service contracts, allowing a virtual enterprise to be formed around a given (set of) contracts and with a group of logistic actors that is willing to provide and accept the complementary role(s) defined by these contracts. By registering, actors guarantee to have the skills that are needed and agree on the interaction protocol that is part of the
contract\(^2\). As a consequence, they become interchangeable and can (re)group at very short notice to form an ad-hoc virtual enterprise to service a particular client demand. The physical location of an actor becomes merely an 'attribute' that may be relevant for one type of service or client, and totally irrelevant for another. Furthermore, actors that fail to do what they promised can be replaced by other actors that meet the same profile. It is just because of this agility that special consideration is given to the issue of actor responsibilities: a logistic contract is not only a technical tool for designers and implementors, it is a legal statement at the same time, stating what parties are required to do after they have accepted a contract.

3.1.3 The design method

Designing a system around logistic contracts constitutes a new approach to designing logistic systems. Whereas usual (Object Oriented) design methods would identify the object classes involved and define their interfaces, Actor Interaction Modeling takes that design method one step further by not only defining the object's interfaces but also adding a protocol to specify how those interfaces should be used. Just like an actor interface provides an abstract view of the tasks that can be performed by that actor, the protocol provides an abstract view of the (lower level) mechanics of the interaction by defining the ordering constraints that determine when these tasks can be executed. Actor Interaction Modeling transforms the specification of object interfaces into the design of a contracting layer that specifies the rights and obligations of the various parties that engage in the contract. The business logic of each actor remains hidden from the other parties. The latter aspect is especially important because actors may reside at geographically different locations and will have to be maintained independently from the other actors; the granularity of upgrading is smaller than the system as a whole, and existing parts will have to coexist and cooperate with newly developed parts (see also the foreword of [CDK93]).

The main difference with traditional software specification techniques is that the latter specify the interfaces of the components (programs, packages, objects, etc.) involved, but leave it to the programmer(s) to ensure that these interfaces are used properly. Even though some techniques and languages allow the designer/programmer to describe assumptions that must hold when a task is started (the preconditions), or that are guaranteed to hold when the task terminates (the postconditions)\(^3\), these conditions are quite abstract and do not necessarily offer any suggestion as to how to implement the system in a way that meets them. In existing software engineering literature, specifying such constraints on the inputs and outputs of routines is referred to as ‘design by contract’ (see e.g. [LG86] and [Mey97]). Logistic service contracts are a logical extension of this ‘design by contract’ paradigm, allowing a designer to specify both logical and temporal conditions (see also sections 3.1.4

\(^2\)Of course, there is no guarantee that the way in which the skills are implemented by the actor conforms to the intention of the contract! This issue can only be resolved by adopting the contract as a legal document that implies punitive measures if a party does not properly implement the skills it has promised.

\(^3\)In addition, there may be (invariants) that describe the assumptions that must hold at any time (or at least at those times that correspond to an observable state of the components involved).
and 5.13). Though specifically targeted to the logistics environment, the ideas may also be applied to the wider field of software engineering. More specifically, contracts might be used to connect the different layers of any so-called multi-tiered\textsuperscript{*} system.

Two aspects of Actor Interaction Modeling are especially important for designers and prospective actors: the first one is the language in which to specify service contracts; the second one is the mechanism used to find possible actors to cooperate with. The first aspect, in particular the requirements for such a language, is discussed in section 3.1.4 (the new language, called LINC, will be presented in detail in part II of this thesis). The second aspect will be discussed in some depth in part IV.

### 3.1.4 The language: LINC

In the foregoing we have argued that the challenges of New Logistics can be met by means of logistic contracting and by a new design method that is based upon modeling the interaction between logistic actors (besides the more traditional definition of the actor interfaces, that is). Like TrAceEs and FoReEs, LINC is intended as a design method that is to be supported by a software environment in which to develop and operate logistic systems\textsuperscript{4}. As a design method, LINC provides a language (carrying the same name as the method itself) to specify the interactions between two or more logistic actors that intend to achieve a certain goal together; the name is derived from Logistic Interaction Contract(ing)\textsuperscript{5}. The development environment provides a software platform for the implementation of logistic actors. This platform consists of a number of basic building blocks that are necessary to validate the interaction between the actors and to search for actors that could become partners in the virtual enterprise to be established. In conclusion, three roles can be distinguished for a LINC:

- it forms a starting point for the implementation of logistic actors,
- it is used to enforce the rules of communication between them,
- it is used as a means to find partners to form a virtual enterprise.

It is this specific combination of roles that is intended to achieve the aforementioned requirements: agility, flexibility, robustness, ease of development and maintenance. These roles also lead to the requirements for the LINC language and platform:

- The language must be usable as a specification method for logistic systems. It has to support the notion of ‘design by contract’ by allowing the logistic engineer to

\textsuperscript{4}The LINC layer of AgileFrames is only concerned with logistic control; traffic control and machine control are taken for granted and assumed to be used to implement the skills that are offered by logistic actors.

\textsuperscript{5}We will sometimes use the same name for an interaction contract (both model and instance, see section 3.3.3) too and refer to such a contract as a LINC.
define the skills of each cooperating party as well as the proper way to make use of these skills. As design documents, logistic contracts can be used by implementors to create logistic actors that conform to other parties’ expectations. This use of logistic contracts extends to the maintenance stage of logistic systems by defining the constraints that must be obeyed when the functionality of the logistic actors has to change over time.

- Apart from being design documents, logistic contracts are also intended to enhance reliability and safety of a logistic system when it is in operation. As a specification of the interaction dynamics, contracts must be usable to verify the interaction and to guarantee that every interaction conforms to the contract.

- Logistic contracts are not only meant to serve as design documents, but also to function as legal documents: they will specify the responsibilities and obligations of the parties that have agreed to execute the contract. Since they are used to verify the interactions taking place when the contract is executed, they can also be used to record information about these interactions. This information may then be used in case something went wrong and some party is to be held responsible.

- Similarly, the information being recorded should be usable for the purpose of tracking and tracking by one of the executing actors, and possibly for billing after the cooperation has terminated. It should also be usable by selected parties to track the operation for other purposes, i.e., customs, taxes, safety boards, etc.

- Logistic contracts are also intended to play a role that is new and will become pervasive in the logistics field: that of locating, contacting, and cooperating with possible business partners in a safe and reliable way. The logistics world is a global market in which different parties are offering and/or searching services of many kinds, so it is a necessity for logistic actors to make their presence known in order to find suitable jobs. Since actors can be characterized by the contracts that they support, they can use these contracts as a bill-boards to attract potential partners.

The following sections will deal with logistic contracting in more detail, and lay the basis for the design of a logistic contracting language.

3.2 Logistic actors

The concept of a logistic actor is an important one; together with the concept of a logistic contract it forms the basis of the LiNC platform. In section 3.2.1 we will show how the idea of an actor is related to the physical world of real logistic operations. Section 3.2.2 describes the visible part of an actor, i.e., its interface. Section 3.2.3 will then define some of the basic capabilities that every logistic actor must have. Finally, section 3.2.4 explains how actors may delegate part of their tasks to subcontractors.
3.2.1 The LinC virtual world

One of the main assumptions made by AgileFrames is that the parties that play a role in logistics operations do not only exist in the real (i.e., physical) world, but also in a virtual world (the LinC community) that mimics the physical world to a certain extent [ELL00, Eve00]. This assumption is obvious: if real world actors want to take advantage of the flexibility of speed of computer networks, they have to present themselves to this virtual world in the form of computer applications having a well-defined functionality. These software applications are what we call (virtual) logistic actors. Nothing is assumed about the parties that can join the LinC community, except that they must all have this virtual alter-ego. Logistic actors can represent parties as large as a world-spanning transportation company or as small as an individual parcel equipped with an electronic transponder chip that monitors a package's state or location. The middle ground in this range is taken by e.g., container-ships, quay cranes, and pallet trucks (see also [Eve00]).

Another assumption made by AgileFrames (see also section 1.4) is that the LinC community is a basically anarchist society of logistic actors that expose themselves to this 'cyberspace' by advertising the facilities they can offer. By searching the cyberspace for providers of needed services, actors can form a 'task force' around the goal of executing a coordinated set of services. Obviously, finding suitable providers would be impossible unless we had an accurate and formal description of the provided services. At the very least, such a service description would have to list the capabilities or skills of the providers in question (see section 3.3.1). In order to interact with an actor in a reliable way, though, knowing the actor's interface is often not enough: most actors want to impose some restrictions onto the order in which their skills may be invoked on behalf of clients. Such restrictions can be specified in the form of a usage protocol. Together, the lists of actor skills and the usage protocol form the core of a logistic contract. Below, we will take a closer look at the actor interface; refer to section 3.3.1 for a more detailed presentation of protocols and the reasons behind them.

3.2.2 The actor interface

Every logistic actor presents its own interface as the provider of a certain service to the outside world. This interface is defined as the set of skills that the actor can offer to its possible clients. During execution of the logistic contract, clients may then ask the provider to perform the tasks that are associated with these skills. Whereas in the foregoing we used the very generic wording of 'software application' to describe a logistic actor, we can be more specific now: an actor is in fact a software object (in the O-O meaning of the word) implementing the service interfaces that are prescribed by the contracts that it supports. Readers familiar with modern Object-Oriented programming languages like Java and C# may notice the similarity to the interface concept as defined by those languages; this is not merely a coincidence, of course: although the service interfaces that are part of service

\[6\] Indeed, an actor may support (i.e., be a party in) any number of different service contracts.
contracts offer some extra facilities (most notably method pre- and postconditions, see section 5.12.1), they are directly mapped onto the interfaces offered by the underlying implementation language.

It is important to note that there is no fundamental difference between providers and clients: clients too may be asked to contribute certain skills to the execution of a contract. Although this effectively turns every logistic actor into a kind of entity that is both a provider and a client, we will continue to speak of providers and clients of 2-party service contracts. This is because there will often be a qualitative difference between the skills contributed by either party, and hence a distinctive provider/client asymmetry.

### 3.2.3 Basic actor capabilities

Besides the skills that are specific for a given actor type, every logistic actor requires a number of basic capabilities that are defined and needed by the platform itself. E.g., each actor needs to be able to present it’s contact information (name, address, phone numbers, e-mail address, URL, etc.) and credentials. Furthermore, it needs to know how to reach the other actors that participate in the logistic contract, how to access the global data used or maintained by the contract, and how to start and terminate the contract (further capabilities will likely be added when the LiNC platform matures). As we will see in section 9.3, an actor obtains all these basic capabilities by declaring itself an heir of the LincBasicActor class. Obviously, in order to be able to participate in the execution of a particular service contract, these basic capabilities have to be complemented by the specific skills required by the contract.

### 3.2.4 Delegation and subcontracting

When actors commit to their clients to provide a certain service, this does not necessarily mean that they are on their own in executing these offered services. Providers of logistic services are free to delegate all or part of their tasks to third parties by becoming clients of other service contracts themselves. When a provider chooses to delegate work to sub-providers, it will establish a regular service contract with those providers, playing the role of their client exactly like its own client has established a contractual relation with itself. Figure 3.1 shows a provider that delegates some of its work to subcontractors. The main client is shown at the left; the inner ellipses in the main provider (shown at the center) denote ‘embedded clients’ for the services that are subcontracted to the two providers shown at the right. The various provider-client relations are as usual; they specify the interactions between individual pairs of logistic actors. Usually parties can delegate without telling the world that they are actually doing so: delegation is part of the inner workings of an actor, and usually hidden from the rest of the world.

There are circumstances, though, that clients want to know if some part of the process is being delegated. In these cases it is necessary to expose part of the inner workings of a
logistic actor to the outside world: process blocks (see section 5.13.12) provide such a partial view to an actor's inner workings. One possible reason to expose a party's subcontractors to its clients is to enable clients to control the provider's subcontracting behaviour. E.g., a client might have reasons of quality control, or even legal or political reasons for wanting to put restrictions on the actors that can be used as subcontractors. This can be done statically, by specifying the subcontracting actors as additional parties in a multi-party contract (section 5.16), or dynamically, by passing them as arguments to the provider in a task invocation message (see section 5.13.1).

Earlier publications on AgileFrames (e.g. [ELL00]) introduced a special kind of contract (a delegation contract), to show the relations between a provider's own service and any subcontracted services. Delegation contracts have not been included in the LiNC language; their functionality has been replaced by the above mechanism.

### 3.3 Logistic contracts

Logistic actors expose themselves to the AgileFrames virtual world by presenting their interfaces and by specifying the corresponding usage protocols. Although it would be possible in principle to specify actors as stand-alone entities, these usage protocols offer a better approach because they define the actors as interacting entities. The central idea of LiNC is to use logistic contracts as a means of specifying the interaction (e.g. communication) between logistic actors in a specially designed contracting language. This idea is based upon a number of observations:

- Logistic contracts can double as design documents, and provide a (partial) specification of a logistic system that is usable both during the design and during the maintenance stage of the system.

  Systems in which communication between actors is intertwined with their business logic are more difficult to design and maintain than systems that separate those concerns. Separating communication (interaction) from business logic gives implementors clear guidelines of what to implement, and leaves them free to decide how to implement the desired functionality. Since logistic contracts not only define the interfaces
3.3. Logistic contracts

but add important usage information, implementing the actor functionality is made easier. Also, the logistic contract documents the constraints to obey when changing existing systems and offers strong support to future implementors that have to maintain or extend the system. Centering the logistic design around the interaction between actors makes the resulting system more robust against inevitable updates of the individual actors, and often the actors can be enhanced or extended while keeping the contract intact.

- Contracts can function as legal documents that specify the responsibilities and obligations of the parties that have agreed to execute the contract.

This aspect is especially important if logistic actors are unknown to each other or plan to do business with each other on a temporary basis, as will be the case in the context of AgileFrames. Whereas parties in more traditional logistic operations usually rely on personal relations between the people that make up those parties, modern Internet-based businesses are sometimes less well known to their prospective partners. Special measures should therefore be taken to ensure integrity of such ad-hoc logistic operations, and safeguard the operation against parties that suddenly disappear or stop working, or that start to behave erratically. Even though a contract cannot prevent a party from causing such trouble (more about this under the next point), the least it can do is form a solid basis for legal action and/or financial compensation claims.

- Logistic contracts can be used to verify the interaction taking place between actors while the contracts are being executed.

This observation is related to the previous one, but instead of being primarily concerned with specifying correct behaviour of the actors, the main concern here is enforcing correct behaviour and/or taking measures after possible misbehaviour from the side of one or more parties. Now that contracts provide us with a formal specification of the interaction/communication between logistic actors, we are able to actually verify the interactions while they are taking place. Obviously, contracts cannot force an actor to act in a well-behaved way; what they can do, however, is protect the remaining actors from another actor’s possible misconduct. Since the system only allows interactions that conform to the usage protocol, the observable state of the system as a whole is always valid. The system is therefore robust against disruptions, and can only proceed in a well-defined way. There is one possible exception to the foregoing, though: both an actor and the protocol verifier may throw a so-called ProcessError exception (see section 5.12.1), meaning that things went wrong beyond recovery. If this happens, parties will usually fall back to the last resort: legal action. In doing so, they can rely on the legal status of the contract (see above), and on the reports generated during the contract’s execution (see below).

- Logistic contracts provide insight into the details of their execution, both while they are running and after they have terminated.

The issue here is monitoring and recording the observable state of the actors that
are interacting under control of the contract. The usage protocol that is part of each contract imposes a certain semantic structure onto the interactions that take place between the various actors. The structure is called semantic because it reveals the intentions (or at least some part of them) behind the interactions that take place, and corresponds directly to the core issues addressed by the logistic design that is embodied by the contract. In order to check all interactions against the usage protocol (see also section 3.3.1), the protocol verifier (see section 9.2.2) has to assign each interaction its proper place into this semantic structure. As a useful side-effect, the resulting semantically structured data is available both when the contract is in execution and afterwards, after it has terminated. Because the contract has as a legal status that is accepted by all parties, and since the structure corresponds directly and provably to the factual interactions that took place during execution, it can be used to resolve disputes about the contract’s execution; the information will usually be archived. Furthermore, the recorded data may be used to generate high level reports about the execution of the contract.

- Logistic contracts provide a mechanism for interaction rehearsal and/or partial simulation.

Logistic contracts provide a detailed description of the actors and the interactions between them, so it should be possible to use them to automatically generate ‘dummy’ implementations for these actors. The dummy actors would do nothing real, but still exhibit a communication pattern that conforms to the usage protocol. Any third party that is interested in becoming a supplier or client of a given contract could then test its own implementation with these dummy actors; this is what we call ‘contract rehearsal’ or ‘interaction rehearsal’. A variation on this theme is the possibility to replace real actors by actors that perform a simulation of the intended tasks.

- Logistic contracts can be used to locate suitable business partners in the first place.

The AgileFrames platform does not only define contracts as a means of guarding communication between actors, it also intends to use them as a means to establish groups of actors that join forces to execute a certain goal. This group-forming process is a necessary prerequisite to get a logistic operation going. In ‘traditional’ distributed applications, lookup- or discovery services (see section 8.6) would be used by a party to search the virtual world for the software objects representing the needed actors; the requested actor’s interface would be used as the search criterion. In AgileFrames the mechanism is similar, but instead of looking for an implemented interface an actor would search for parties that support a given logistic contract in a given role. Given a ‘database’ of logistic contracts (the Logistic Contract Space, see section 3.4), parties can search for logistic actors that are offering to play a role in each contract. Depending on the attributes of the actors that are found, the one actor that suits best can then be selected and contacted.

While the above remarks are mostly about the benefits that may come from using logistic contracts, they don’t go into any detail about what has to be specified in a contract in order
3.3 Logistic contracts

to achieve all these noble goals. The next section (section 3.3.1) will provide a short introduction to the main functional parts of a contract (the fine details can of course be found in part II of this thesis). In section 3.3.2 we will clarify the difference between contracts (which are descriptive in nature) with (declarative) programs (which are prescriptive). Finally, in section 3.3.3, we will elaborate on the life cycle of logistic contracts.

3.3.1 Service contracts

When talking about logistic contracts, we have to distinguish between the literal text of a contract, i.e., the model contract, and the specific instantiation of a model contract by a group of logistic actors that agree to execute the contract together. In the remainder of this thesis, we will just use the word 'contract' to refer to either meaning: the intended meaning will always be clear from the context.

Since logistic contracts describe some important aspects of logistic services, we usually refer to them as service contracts. There are two kinds of service contract, 2-party and multi-party (n-party) service contracts. 2-party contracts are the more common form, but in fact they are just a special case of multi-party contracts. If we use the term ‘service contract’ without any further qualification, it is assumed to be a 2-party contract. Every service contract specifies the interaction between two autonomous logistic actors. One of these actors, the provider, provides a service to the other actor, the client. A multi-party contract specifies the interaction between at least two contract parties. This allows the logistic designer to define additional roles in the contract, e.g. a patron or a guardian. By the same mechanism it is possible to assign observer roles to external parties, e.g., an insurance company or the tax office. Multi-party contracts generalize the idea of a 2-party contract by specifying the pairwise interactions between m clients and n providers. As the same actor may play the role of a client in one interaction and of a provider in another, these roles are not distinguished anymore, and the protocol is generalized to specify the interaction patterns between all logistic actors at once.

In order to support the search process that is to locate suitable actors and contracts, every service contract specifies the name of the service as well as the service domain it belongs to (see section 5.6). Furthermore, it specifies the types of all actors that play a role in the contract. These actors act as parameters that must be provided with proper values when the contract is instantiated (see section 3.3.3): only an actor whose type matches an actor type in a contract can play the corresponding role in that contract. The main sections of a service contract describe the skills the actors have to have in order to participate in the execution of the contract (see section 3.2.2) and the interaction protocol that they have to adhere to while doing so (see the beginning of section 3.3). Every service contract ends with a specification of the data to be recorded during its execution (see section 5.14). Both

---

7 Sometimes the client is referred to as the principal and the provider as the contractor or performer.
8 The careful reader will notice that there may be some redundancy between the actor's type and the specification of its skills; this is intentional, as an actor might support more than one service contract and implement different and possibly overlapping interfaces.
skills and protocols will be explained in more detail below; the complete information can be found in the LiNC language description (sections 5.12 and 5.13, respectively).

Skills

A logistic actor wouldn't be very useful unless it provided or needed some skills. A skill is a capability provided by an actor (the provider); it can be called upon by other actors (the clients). As we already saw in section 3.2, the collection of skills that can be provided by an actor is called the interface of that actor. A model contract specifies the interface of every actor that plays a role in it, and logistic actors that want to play a role in a contract have to implement the interface corresponding to that role. When a skill of an actor is being executed on behalf of a client, we speak of a task. A certain skill can be invoked more than once; the resulting tasks may even be executed concurrently. Since the difference between skills and tasks is always clear from the context, we will often simply ignore it and refer to both as tasks (even in the LiNC language itself). Apart from the skills that are specified in the service contract, every actor needs some basic skills in order to be able to participate at all. These basic skills are obtained by means of inheritance from the LincBasicActor class (see also section 3.2.3).

Protocols

A service contract specifies the contractual commitments between both parties, the major part of the contract being the protocol. A nice definition of protocols can be found in [FF84] (page 63):

“A protocol is an organizational framework for communication: a mapping between the ordering of symbols and their interpretation.”

The single most important aspect of a protocol is the fact that it structures communication in a way that is agreed upon and known to all parties involved (see also [OTG02]). In particular, the protocol that is specified in a service contract (henceforth called the service protocol) specifies the interaction pattern between both parties. This interaction pattern conveys important usage information that cannot easily be given using conventional software interfaces, even if these interfaces include preconditions, postconditions, and assertions (see also section 3.2.2). Service contracts usually specify the interaction patterns both under regular and under foreseeable exceptional circumstances. Because exceptional situations are more likely to occur than their name would suggest or contract writers would

---

9Protocols are commonplace in hardware design, where communication channels (be it hardware buses, cables, etc.) are used to transfer different chunks of information at different moments. They also play an all-important role in the specification of networking software, where they are arranged in layers to form ever higher levels of interaction. In fact, it is a little surprising that they do not already play such a role in other areas where objects (in the O-O sense) intend to communicate.
3.3. Logistic contracts

hope, the specification of what should happen under exceptional circumstances might even
be the most important part of the contract!

Service objects

Almost every service contract involves the handling of something, mostly the handling of
goods or materials; less frequently, a service might involve the handling of livestock, possibly
even humans. The handled entities (alive or not) are usually affected by the service, i.e.,
their location and/or other attributes will probably change during execution of the service.
A contract’s service objects (see section 5.8) are used to refer to those entities: through
the service object(s), contract parties have access to all that is known about them. Service
objects are defined by types that must be defined in the Logistic Contract Space*.

When seen from the actors of a service the service objects are read-only; it is for this
reason that they can be used as an accurate description of the entities’ status at any time.
Obviously, service objects must somehow be able of being updated, or otherwise they could
not be used to play any useful role whatsoever. To control the updating of service objects,
the only ‘party’ that is allowed by the current language to update their values is the service
contract itself. This is done by allowing for update statements (see section 5.13.11) in the
service protocol\(^{10}\). Assuming that these updates take place at appropriate times to reflect
the actual changes that were made to the service objects, the latter can be used as an
accurate description of the represented entities at all times.

To conclude this discussion, it should be mentioned that the service objects themselves are
left unspecified by the LINC language: a service contract just lists their name and type
(a Java class name, see section 5.8). A service object could therefore be anything from
a simple object to a complete database. The aforementioned update statements however,
make use of knowledge about the service objects and the operations that are allowed on
them. The LINC compiler therefore needs access to the class files corresponding to the
service object types.

3.3.2 Why service contracts are not programs

It is important to realize that a service contract is much unlike a program in the usual
sense, even though many of the constructs that will be described in part II may look similar
to programming language constructs. Unlike regular programs written in imperative (or
functional, for that matter) languages, contracts do not specify that something is to be
done or computed, at least not by the contract itself. Instead, the actors involved in the
contract are interacting while they are executing their own (invisible) business processes.
The contract is there to verify that their interaction attempts do not violate what was
agreed upon. One could compare a contract to a grammar, and the factual interaction

\(^{10}\) Apart from being used in update statements, service objects may also be used in assert statements (see
section 5.13.10) that state the conditions assumed to be true at a certain point of execution.
to a sentence uttered in the language described by the grammar. The grammar would mark ungrammatical sentences as such; so does the contract. The contract is really just a declarative specification of valid interaction patterns. It does not direct the contract parties in any way; instead, it tells them when they attempt to do something illegal and thereby protects the other parties from being confused.

Another idea that is central to contracts, is that they are only a partial description of what actually happens during their execution, and that their intention is to leave as much freedom as possible to the business logic to fill in the necessary details. Although this freedom is partly achieved by leaving the details of each task undescribed by the contract, the contract writer should also help to ensure it by not using constructs that restrict the implementation unnecessarily. For example, when two constructs (e.g., task invocations, see section 5.13.1) are independent from each other, the contract should not force them to be executed sequentially, or even in order. So again the contract is unlike a program in the usual sense, as it does not specify a specific execution flow.

Related to the foregoing is a contract's 'open-endedness': even though a service contract describes the expected interaction between the logistic actors, reality often takes over by putting the actors in a situation that was not foreseen. No predefined contract can ever be complete in the face of such exceptional situations. Usually, unexpected situations result in a task being ended by an exception. Some exceptions may be foreseen; these can still be handled under direct control of the service contract. This can be done by inclusion of the handler protocol in the contract or by instantiating a sub-contract to handle the problematic situation. Other exceptions may not be foreseen; these will require the actors to rely on improvised interaction between parties or even need external support by human actors.\footnote{It is even possible to design or select a new service contract and instantiate it right away while handling the exception!} Service contracts must therefore be able to fall back to 'external' procedures and 'protocol-less' operation. The ultimate fall-back, however, is to notify the other parties and wrap up the contract.

### 3.3.3 Life cycle of service contracts

Service contracts are similar to programs in one way, though: they have a life cycle that consists of a document phase, an execution phase, and a result phase. In the document phase a contract is a piece of text to be agreed upon by the prospective contract parties; this is why we called it a model contract in section 3.3.1. The representation of this text is irrelevant: it might take the form of a HTML document that can be viewed on-line, a regular ASCII file, the Java source code that is generated by the LINC compiler, or the set of class files resulting from compiling that code. All these 'documents' represent the same thing: the original contract.

The second stage is the execution phase of the contract; the phase in which it exists as an instantiation of the model contract. The term 'execution' may sound a bit contradictory
3.4. The Logistic Contract Space

after what was said in the previous section, but it is not: even though a contract in itself does not represent any executable logic, all communication between the contract parties is monitored and verified according to the contents of the contract. If the protocol should be violated by a certain party, that party will be informed so it can correct the situation (the other parties won’t even notice that some problem occurs). In this sense a contract also has an execution phase, i.e., a phase in which it is active and performing the task it was designed for.

The final stage is the result phase or report phase. As will be shown in part II (section 5.14), contracts in execution create a journal tree or shortly, a journal, containing all the details of the interactions that take place. The journal tree, which can be saved in the form of an XML document (see section 5.14) may be used to generate one or more reports afterwards, so the last stage of a contract’s life can be compared to the ‘output’ of a regular computer program. This output serves an important goal: it is a legal document that provides a record of the interaction, and as such it may be used for e.g., billing, auditing, suing, etc. Or it may simply be stored in an archive.

3.4 The Logistic Contract Space

Logistic contracts are maintained in the so-called Logistic Contract Space, a globally distributed hierarchical name space which is available from anywhere. Clients may use the Logistic Contract Space to look for providers advertising services of some kind, while providers may use it to look for prospective clients for their services (see also section 9.1). The Logistic Contract Space is a structured name space containing separate subspaces for different service domains as well as a public space for contracts that don’t fit well in a specific domain. Figure 3.2 shows what the Logistic Contract Space might look like. Every service provider, client, or contract has its own unique place in the Logistic Contract Space; this place is determined by the actor type or the contract’s domain, respectively (see also section 3.3.1). The Logistic Contract Space forms a framework in which logistic actors may advertise themselves by uploading (part of) their code in addition to any required supporting classes; while in most cases it will be the provider that is defining a service, there is nothing that prevents clients to do the same. Apart from logistic actors, the Logistic Contract Space will contain source text of logistic contracts as well as the code that resulted from their compilation. In order to guarantee the integrity of contracts and code, the various domains of the Logistic Contract Space may want to enforce some kind of authentication and access control. This is an issue to be discussed and implemented by AgileFrames user communities and standards organizations.
Figure 3.2: Sample Logistic Contract Space
Chapter 4

The design of a contracting language

As will be clear from the previous sections, the primary goal of our research was to design a language for describing logistic actors and their interactions, and to demonstrate that logistic systems can be effectively described by means of this language. A secondary goal was to develop a (prototype) compiler for the language and to show that the interaction between actors can be effectively verified. In this chapter, we will define a set of requirements for the design of our logistic contracting language and its implementation; in doing so, we will draw upon the explanations of the function and purpose of logistic actors (see section 3.2) and logistic contracts (section 3.3). First, section 4.1 will present the requirements for the design of a (logistic) contracting language. Following that, section section 4.2 will present the requirements for an implementation of the language.

4.1 Requirements for the LinC design

Languages are used to describe something, and the LinC language is no exception. When a new language has to be designed, it is important to define the audience to be addressed and the problem to be solved by using the language. Since AgileFrames is about designing and implementing logistic systems, the intended audience is clear: the language is to be used by logistic engineers who need to define the systems they want to build. But designing a logistic system is not the same as designing and implementing software: that task is best left to software engineers, i.e., people who do not necessarily understand what is best from a logistic viewpoint. As a consequence, the language must also be understood by the latter caste. Actually, the software designers and programmers should be at least as capable in understanding the language as the logistic designers are! With this audience and purpose in mind, we can now define a minimum set of requirements that the language has to fulfill. The requirements can be divided into two separate categories: requirements that are specific to the intended purpose of the language, i.e., the design of logistic systems, and requirements that apply to the design of any formal specification of programming language. The next section will start with the first category.
4.1.1 Domain-specific requirements

The logistic contracting language, which was named LINC after the AgileFrames layer that it was to be part of, had to be based on the ideas we outlined in the preceding sections. In particular, it had to satisfy the following domain-specific requirements:

- provide a complete description of the interfaces of all actors that will play a role in the logistic service.

Logistic contracts are publicly available documents that can be used by any person or company that intends to develop software to implement logistic actors. When doing so, third parties are expected to use the LINC compiler to generate the interfaces to be implemented. These interfaces cannot be extended, though, since all communication has to pass through the protocol verifier, which does not accept any communication that was not in the original contract\footnote{This limitation is intentional, since we want to be able to have a 100% accurate description of the interaction. Later versions of the language might extend the language with some form of contract inheritance in which actor interfaces as well as usage protocols can be extended to cover new interaction patterns.}. As a consequence, the interface has to be complete, i.e., specify all skills of the contract parties in every detail.

Even though most regular programming languages do not have an elaborate assertion mechanism (Java 1.4 assertions do not qualify as such, whereas the Eiffel mechanism does), we also want the interface to state every condition that is necessary to ensure proper usage. Since the interfaces are supplemented by a usage protocol, it is only necessary to state those conditions that are not implied by the protocol.

- provide an adequate description of the runtime interaction between all actors involved in the contract.

In section 3.3 we have stressed that the protocol provides us with a semantic structuring of the communication taking place. This remark presupposes that the protocol does indeed provide us with such a structured view of the interaction! Not being a programming language (see section 3.3.2), then what is it that the language has to do? As we said above, the protocol is like a grammar describing the dialogue between the actors. In order to be an adequate description of this dialogue, the protocol has to be able to capture all the relevant aspects of the communication that results from executing the logistic actor's business code. It is this observation that provides the necessary clue: the interactions are generated by actors that are programmed using the common control structures of an implementation language, and our language therefore has to provide structuring mechanisms that somehow parallel the control structures of programming languages, albeit at a higher abstraction level! I.e., the protocol description should at least be able to express sequential and concurrent execution, as well as some form of repetition and choice. Of course, the protocol writer should not try to duplicate what should be left to the actor implementors; this is one reason why we chose not to include explicit control of e.g. the repetition and choice interaction patterns.
4.1. Requirements for the LinC design

- allow formal verification of the actual runtime interaction and protect parties from erroneous behaviour of other parties.

As was explained in section 3.3.3, logistic contracts have an ‘execution’ stage in which they are used to verify all interactions taking place between the actors of the contract. The protocol must therefore be formal enough to allow all task invocations (the only elementary interaction) to be matched against the protocol structure and be attributed a logical place within that structure. Notice that we don’t say ‘unique’ even though the current implementation indeed requires that every valid interaction can be attributed to one and only one position in the protocol structure. In other words, it is currently not allowed for a protocol to be ambiguous. Although as yet we do not see any advantage to allowing protocols that are temporarily ambiguous, we might encounter good reasons later to allow for them.

- allow for the handling of ‘regular’ exceptional situations within the scope of the contract.

The skills that are specified in an actor’s interface correspond directly to methods that can be invoked on the object that implements the actor. Since methods may either terminate normally or by throwing an exception, the same is true for skills. As a result of this, a task invocation (see section 5.13.1) may end with an exception too. The party that tried to invoke the failing task will then have to do something special in order to handle the exception before it can continue executing the contract. Usually this will mean that some further interaction is needed in order to determine the cause of the problem and retry the failing task. The protocol verifier therefore has to detect the exception and allow for its handling, and the language must be able to express exception handling as well. The reason we called these exceptions ‘regular’, is that they are still handled within the protocol; and in order to include their handling, they must be foreseeable. Exceptions that cannot be foreseen at all cannot be handled under control of the contract; these are the topic of the next remark.

- allow parties to escape from the contract in case of situations that cannot be handled within the scope of the contract.

The previous remark covered exceptional situations that were known to happen every now and then; while being exceptional, such situations can hardly be called unexpected. Really unexpected situations cannot be described, of course, and if they happen they cannot be handled under control of the protocol. In order to deal with them, the language will have to support one or more exceptions that may either be handled outside the contract or end the entire interaction by terminating the contract. In the latter case, the party throwing the exception will have to face a compensation claim. As all communication is logged (see below), this claim may be based upon the information obtained while the contract was still active.

- allow for structured logging of the all interactions taking place.

The language has to support interaction logging in a way that makes all relevant information available, both during the contract’s execution and after it has terminated
(see also section 3.3.3). More specifically, the logging should be done in a way that captures the semantic structure of the interaction as reflected by the protocol. Since not all language constructs will be equally relevant, it would be best if the contract designer can control the level of logging to be used, i.e., which language constructs are relevant to be shown in the resulting log. In order to allow for wide application, the data should be logged in some way that allows it to be used as input for any other application (XML may come to mind here).

- allow actors to find and contact possible partners in the execution of a contract.

As the contract is itself a description of all the actors’ types, interfaces, and the correspondent usage protocol, this requirement is basically fulfilled by having a contract in the first place. The only language requirement is that the actor types are specified in a way that allows them to be used as a search criterion. Different discovery mechanisms (see section 8.6.1) might lead to different requirements; ultimately, the contract text itself may be used as a search key.

### 4.1.2 General language design requirements

Apart from these domain-specific remarks, there are some general aspects that apply to the design of any formal language. Designing a language entails making many important decisions. E.g., it involves creating a syntactic and semantic framework, deciding what to include and what to leave out, deciding on the level of syntactic redundancy, choosing the ‘best’ constructs for the things to be expressed, etc. The following list presents some aspects that we find especially important (the list could of course be much longer, and as usual in matters like these, some points are open for discussion):

- A language should introduce a minimum of new mechanisms; instead it should draw upon the user's existing knowledge of languages. New constructs should serve to introduce new mechanisms to the language (when unavoidable), not to create a new language!

- A language should address similar problems in similar ways, and distinct problems in distinct ways. Using a single mechanism to accomplish different goals adds to confusion and might be a source of misconceptions.

- Language structures should be orthogonal: it should be possible to combine independent structures in many ways. The language must be designed in such a way that all those combinations make sense.

- A specification language should force the user to abstract from irrelevant detail without compromising his ability to describe the necessary execution details. Neither should it hamper the implementor’s ability to create an efficient compiler.

- A language should not define any mechanisms that can only be understood with detailed knowledge of the underlying implementation; its structures should reflect the
expected and readily observable behaviour of the implementation, and nothing should happen ‘auto-magically’.

These guidelines were kept in mind when designing the LINC language. Although many of them will sound familiar to language designers, they are included here as an explanation of why the language changed a lot compared to earlier versions as described in [ELL99, ELL00].

No effort is made to compare the details of the language defined here to the corresponding mechanisms in the older publications; the interested reader can refer to them to see the differences in detail. While the mechanisms were all there (with only few exceptions), the form has been changed and been made to comply with the above requirements. The goal was to be as clear and expressive as possible, and to create a language that would help the writers of contracts to specify what they want with a minimum of effort.

In addition to the decisions concerning the language itself, one has to decide on the software platform to build upon when creating an implementation, and on the level of security that is considered appropriate for the system. These topics will be briefly discussed below.

4.2 Requirements for a LinC implementation

Compared to the rather extensive list of requirements to be fulfilled by the language design, the implementation requirements are relatively straightforward. Obviously, any implementation will have to realize all the proposed language constructs and mechanisms: at the very least this would mean that the language is fully implemented. Also, the implementation of the discovery mechanism must be flexible enough to locate suitable actors in any geographic area. Part IV will show to what extent these requirements are fulfilled by the current pilot implementation. Apart from these, there are some extra requirements that must be fulfilled if the LINC platform is to be accepted for real-world logistic operations. These requirements are: use of common software standards (see section 4.2) and a sufficient level of security (see section 4.2).

Existing standards

For any system to be successful, it has to be widely available and adhere to existing (preferably free) standards and software. During the last decade, the Java platform has become more and more important, and the number of software products for Java has grown immensely. One such ‘product’ is the Jini platform, which has been designed to handle the interaction of devices (e.g., computers, printers, scanners, cameras, etc.) in a dynamic fashion that allows any device to be (dis)connected to and from the network at any time without causing major disruption. Jini (as envisaged by Sun Microsystems) defines both an API for robust distributed application programming and a specification of how this API should be used by well-behaved applications [AOS+99, Edw00, OW00]. Jini is totally implemented in Java, and builds upon available libraries (like Java RMI). Since Jini, another kind of
Chapter 4. The design of a contracting language

distributed system, called peer-to-peer, has been developed; project JXTA (again initiated by Sun Microsystems) was established to create a well-defined set of protocols for P2P communication [JXTa, OTG02, Wil02]. A reference implementation in Java is available from Sun. The advantage of these platforms is that the software is freely available for non-commercial use and that the corresponding commercial Enterprise Editions have gained wide acceptance in the Java world. Alternative platforms, like Microsoft’s .NET, also look promising, so alternative implementations are possible. Since the JVM and the .NET VM are not compatible, it is currently not possible to create a LINC implementation that allows for cross-platform interaction contracts.

Security issues

Nowadays it is only common sense to devote a lot of attention to security matters. Not only to prevent curious souls from overhearing other parties’ private conversations (and maybe gaining insight in their private business processes as well), but also to prevent less innocent people to try to masquerade for others and try to establish false contracts under their names. Authentication and authorization techniques should therefore be employed to distinguish between trusted and untrusted parties. Another important aspect in relation to virtual enterprises is what is called non-repudiation by [Kay03]: actors should not be able to deny their past actions. While the use of secure communication channels like SSL (Secure Socket Layer) and VPN’s (Virtual Private Networks) is common, these mechanisms are not enough if sensitive operational information has to be stored in intermediate network nodes and/or if a single protection level is not enough. The latter situation occurs if some information is private to some parties in the contract while visible to others. The reader is referred to [Kay03] for more information.

While we certainly acknowledge the importance of security issues, they were largely ignored by our research, which focused on other aspects of the interaction. One security issue that is implicitly address by our research is the aforementioned non-repudiation: service contracts offer a secure and complete way to log all interactions taking place, so it is impossible for a party to deny its actions once they have occurred. See sections 5.14 and 9.2.4 for more information.
Part II

The LinC Language
The second part of this thesis provides a detailed description of the LiNC syntax and semantics. Whereas the syntax is specified in a formal way (using the common EBNF notation), the semantics of each language construct are specified informally using plain English. We have however tried to be as ‘formal’ as possible, and completeness was considered more important than readability. The use of most constructs will be explained by means of a short example. Readers who want to see more detailed examples of the language in use are referred to part III, which contains a large number of contracts that were designed as part of our case studies.

Many sections of this part contain ‘Rationale’ paragraphs explaining the reasons behind a particular syntax and/or semantics. There are two reasons to include such information as an integral part of the language description: not only does it document the design process, it also helps in understanding of those rules that are not self-evident. Readers who just want to know the rules of the language may contact the author to obtain Language Reference in electronic form whose text is identical to chapter 5 except for the omission of these rationale paragraphs.
Chapter 5

The LinC Language Specification

This chapter provides a detailed description of the syntax of the LinC language. The syntax rules are presented using the well known EBNF-notation. Whenever relevant, the function of a construct will be described and a few examples will be given. The scope of these examples is necessarily limited though, and the reader is referred to subsequent chapters for significantly more detailed examples. When trying to understand a language, it often helps to know the reasoning that goes behind the design of each construct. For this reason we have included a short synopsis of this reasoning in sections headed by the caption 'Rationale'. To set them apart from the main body of the text, these sections have been typeset using a smaller font.

5.1 Notation

Even though the notation we will use to describe the LinC syntax is fairly common, the intended audience of this thesis is diverse enough to warrant a short explanation. There are two different ways to describe a syntax (meta-syntaxes) that are widely used: one way is by means of so-called 'railroad diagrams', the other is by means of the (E)BNF notation\(^1\). Since the latter notation is more compact, we will use it in the rest of this chapter. BNF rules are simple rewrite rules, consisting of a lefthand side, a rewrite symbol ('::='), and a righthand side. The lefthand side is a single so-called nonterminal symbol or shortly nonterminal. The righthand side is a sequence of nonterminal and terminal symbols. Symbols belonging to the latter class are to be taken literally. In our notation, terminal symbols will be written in bold and surrounded by single quotes, while nonterminal symbols will be written using regular Roman typeface. The meaning of the rule is that the lefthand-symbol can be rewritten as the sequence of righthand symbols, or vice versa (the rules can be used to generate or to analyse program (or in our case: contract) text). The difference between BNF and EBNF is that EBNF introduces some additional metasyntax to make life easier for the grammar writer: the vertical bar ('|') is used to denote that either the symbol (or sequence of symbols) to its left can be used or the symbol (or sequence) to its right.

\(^1\)BNF stands for Backus-Naur form, named after John Backus and Peter Naur; the leading 'E' stands for 'extended'.
Parentheses might be necessary to ensure proper scope for the ‘|’ symbol, e.g. ‘S ::= A B | C D’ is a shorthand for the two rules ‘S ::= A B’ and ‘S ::= C D’, while ‘S ::= A (B | C) D’ is shorthand for ‘S ::= A B D’ and ‘S ::= A C D’. Additional syntax is used to denote optional constructs and repetition: square brackets (‘[]’) around a (sequence of) symbol(s) denote that the part between the brackets may be omitted; a ‘*’ (Kleene star) behind a construct means that this construct may be repeated zero or more times, and a ‘+’ denotes that the construct may be repeated one or more times.

### 5.2 Keywords

A number of keywords are defined by the LINC language. As usual, these keywords are reserved words, and as such they cannot be used as names. The LINC keywords are:

<table>
<thead>
<tr>
<th>actor*</th>
<th>journal</th>
<th>optional</th>
<th>retry</th>
</tr>
</thead>
<tbody>
<tr>
<td>actors</td>
<td>limit</td>
<td>ordered</td>
<td>select</td>
</tr>
<tr>
<td>and</td>
<td>map</td>
<td>parameters</td>
<td>sequential</td>
</tr>
<tr>
<td>any</td>
<td>maximum</td>
<td>post</td>
<td>service</td>
</tr>
<tr>
<td>assert</td>
<td>minimum</td>
<td>pre</td>
<td>tasks</td>
</tr>
<tr>
<td>client</td>
<td>multiple</td>
<td>private</td>
<td>throws</td>
</tr>
<tr>
<td>concurrent</td>
<td>nested</td>
<td>process</td>
<td>timed</td>
</tr>
<tr>
<td>description</td>
<td>new</td>
<td>protocol</td>
<td>timeout</td>
</tr>
<tr>
<td>domain</td>
<td>none</td>
<td>provider</td>
<td>to</td>
</tr>
<tr>
<td>duration</td>
<td>not</td>
<td>public</td>
<td>try</td>
</tr>
<tr>
<td>end</td>
<td>null</td>
<td>reject</td>
<td>unordered</td>
</tr>
<tr>
<td>exit</td>
<td>objects</td>
<td>report</td>
<td>update</td>
</tr>
<tr>
<td>indexed</td>
<td>of</td>
<td>restart</td>
<td>variables</td>
</tr>
<tr>
<td>indexes</td>
<td>or</td>
<td>resume</td>
<td>when</td>
</tr>
</tbody>
</table>

All keywords are case-insensitive. The convention is to write keywords in lower case. Keywords marked with * are part of the proposed extension for multi-party contracts (see section 5.16).

### 5.3 Names

Names refer to entities of some kind, e.g., to an object, type, task, service contract, service domain, service provider, etc. Names can be simple names or qualified names. A simple name is defined to be equivalent to a Java identifier, while a qualified name is a sequence of simple names separated by periods:

```plaintext
name ::= simple_name | qualified_name
simple_name ::= 'valid Java identifier'
qualified_name ::= simple_name (',', simple_name) *
```
A simple name always refers to some entity that is locally visible, such as an actor name (section 5.16.1), a service object name (section 5.8), a service parameter name (section 5.9), an index variable name (section 5.10), or a temporary variable name (section 5.11). A simple name can also be used to refer to a type (section 5.4) or to a (sub-)contract (section 5.13.2) that is defined outside the current contract but within the same domain (see section 5.6) as the current contract. Qualified names refer to entities that are not locally visible, such as sub-contract names and type names imported from different domains within the Logistic Contract Space. Qualified names must always be specified relative to a root location* of the Logistic Contract Space. As in Java, there is a distinction between the corresponding lower case and upper case letters. When writing contract specifications, it is advised to use the same naming and capitalization style as Java programmers do.

Rationale

Naming and capitalization are part of an adopted style. Unless one has the hope to change the world, it is best to adhere to a style that is in common use on the platform that is used. Currently, the platform is Java/Java RMI/Jini, so the Java conventions were adopted. If LInC were to be re-implemented for a different platform, the conventions for that platform should be followed instead.

In the grammar rules defined in the remainder of this chapter a special convention will be followed to refer to names of a specific kind (e.g., domain names or task names). This convention is that the class name of the referent may be inserted into the nonterminal symbols ‘simple.name’ and ‘qualified.name’. To make this more apparent, the part of the name that refers to the name’s kind will be printed in italic letters. For instance, the nonterminal ‘qualified.domain.name’ will be used to define a domain name that must be a qualified name. Likewise, the name ‘simple.task.name’ will be used to describe a task name that must be a simple name.

5.4 Types

A type specifies a set of values and a set of operations that can be applied to those values. In the current version of the LInC language, there is no way to define new types within the language itself: all types are either builtin into the language or predefined in the Logistic Contract Space:

\[
\text{simple.type ::= name}
\]

All there is known about a type is its name, maybe some constant values of the type, and the operations that can be performed on values of the type. If new user-defined types are needed, the appropriate definitions must be written in some programming or specification language (e.g., Java or CORBAIDL) and uploaded to the Logistic Contract Space.
Chapter 5. The LinC Language Specification

Rationale

Defining types within the LinC language would extend the language considerably, almost to the point where it would become a full-fledged OO-language. Embedding IDL as a type description language into LinC would render the task more manageable, but at the same time impose some undesirable restrictions on the way LinC can be used. In particular, IDL is more restricted with regards to exchanging objects that are not completely described at both sides. Some applications may need the flexibility offered by Java RMI.

Furthermore, one could argue that defining types is beyond the scope of LinC, which focuses on the interactions between logistic actors and should not try to get involved with the contents of the messages exchanged between actors.

Defining types by abstract names achieves both goals at once: complete freedom is offered to implementors (who can use any type they like as long as they put the relevant class files at their proper place) and understanding of a contract is not hampered by implementation details, easing the life of contract readers as well as writers.

All LinC types, even builtin types, are reference types: there are no primitive types like Java `int`, `boolean`, etc. Since such primitive types are not allowed in a service contract, the corresponding wrapper-types must be used instead.

Rationale

Primitive types were dropped because the distinction between primitive types and non-primitive types is considered an implementation detail, and because including them would complicate the semantics of parameter passing.

As mentioned in section 5.3, all types that are defined in the same domain as the current contract can be referred to by simple names. All types outside the current domain must be referred to by fully qualified names that are relative to the root location of the logistic contracting space. Every type name must refer to a Java class file that is part of the package corresponding to the domain and service in the logistic interaction base. It is an error to use a type name for which no corresponding class file or Java source file can be found. If only a Java source file is found, the LinC compiler will create the corresponding class file on the fly.

Depending on the type's usage, there might be additional restrictions: e.g., some contexts require a type to be `serializable`, while some other contexts require it to be `remote`. Such restrictions will be checked by the compiler, and will be documented whenever appropriate. Whenever a variable or task argument is defined to be of some type, all primitive operations of the type are automatically available. Constants of a type must be specified by their qualified names. Examples will be presented where appropriate.
The class files are needed to check if these additional restrictions are met. The LiNC compiler can use them to verify that `Remote` or `Serialized` are indeed implemented. In addition, they are used to validate the expressions that are used in task pre- and postconditions (see also section 5.12.1). Note that the use of class files does not restrict the implementation language to Java, as the number of languages that can be compiled into class files is growing (e.g., Ada95 [JGN]).

Maps

The types described in the foregoing were all defined by classes, external to the LiNC language. There is one special class of types that has not been mentioned yet: map types or shortly maps. Maps are mostly like regular types, but require a little more language support. Map definitions are described by the following rules:

\[
\begin{align*}
\text{map.type} & := \text{map} \text{ index.types \ of \ item.type} \\
\text{index.types} & := ['\text{index.type \ (', '\text{index.type} \ ) \ * \ '}'] \\
\text{index.type} & := \text{simple.type} \\
\text{item.type} & := \text{simple.type}
\end{align*}
\]

Maps are associated with a sequence of index types and a single item type. Any existing type that supports equals () and hashCode () can be used as an index type. There are no restrictions to the item type. Map variables are often used in task invocation messages inside multiple interaction patterns (see sections 5.13.1 and 5.13.9).

Rationale

The reason for introducing maps into the language can indeed be found with multiple interaction patterns (see section 5.13.9). This protocol construct is used to describe iterative behaviour of the contract parties; for each iteration a new instance (branch) of the nested interaction pattern is created to collect the interactions corresponding to that branch. Since most task invocation messages contain variables to capture the values passed between the contract parties, there would be a problem if regular variables were used: the values for the variable would have to be the same in every branch! Obviously this is not the case, and even contrary to the very purpose of the multiple construct. So either we would have to allow for local variables inside the multiple construct, or we would have to use map-like types with one element for every branch. It was decided to use maps instead of local variables. Since multiple-constructs can be nested, we need maps that may have n indices (with n > 0).

5.5 2-Party Service Contracts

A 2-party service contract is a specification of a service in the form of a contract between a service provider and a service client. Because service contracts are mainly about interac-
tion and communication between the parties involved, the language to describe them was
designed to avoid any bias in favour of one of the contract parties and to be as symmet-
rical as possible. As will become clear in section 5.13, this symmetry is especially visible
in the service protocol. All subsequent sections of this chapter assume 2 contract parties.
Multi-party contracts are quite similar though, and will be described in chapter 5.16.

Rationale

Symmetry in the contracts is a prerequisite for the desired symmetry in the way con-
tracts are used: not only may clients look for providers in the Logistic Contract Space,
it is just as well possible for providers to look for clients in need of their services. Ac-
 actually, looking at contracts from a communication oriented viewpoint, it is not at all
relevant who is the client and who the provider. As a result of this symmetry, providers
may ask their clients to perform some task in the same way as clients may ask providers
to do something for them.

A service contract consists of several sections. The first section provides the contract
identity and context, the most important part being the name of the contract. The 'actors'
section describes the actors involved, while the 'objects' section describes the types of
objects acted upon. The remaining sections each describe some aspect that has to be
agreed upon, like e.g. the interaction pattern or protocol.

\[
\text{service.contract} \\
\quad::= \text{service.description} \\
\quad\quad\quad\text{service.actors} \\
\quad\quad\quad\text{service.objects} \\
\quad\quad\quad\text{service.parameters} \\
\quad\quad\quad\text{service.indexes} \\
\quad\quad\quad\text{service.variables} \\
\quad\quad\quad\text{service.tasks} \\
\quad\quad\quad\text{service.protocol} \\
\quad\quad\quad\text{service.journal} \\
\quad\quad\quad\text{service.report}
\]

Each of the contract sections will be described below. Only some small examples will be
included here. Chapters 6 and 7 will provide examples in the form of extensive case studies.
Compilation and execution of contracts will be explained in chapter 9.

5.6 The Service Description

Every contract starts with a section that is called the service description. The service
description provides an identity for the contract, which makes it possible to find and refer
to the service. It is defined as follows:
5.6. The Service Description

```
service.description ::= 'description' service.domain.name service.service.name
```

The service domain and name are defined below.

**The service domain**

The first part of the service description specifies the *domain name*. As explained before, all compiled contracts will become part of the Logistic Contract Space, either in a named domain or in one of the unnamed *public* or *private* domains. Every contract belongs to exactly one domain, whose name must be given following the keyword *domain*. Since the Logistic Contract Space is a hierarchically structured name space, the domain will be denoted by a qualified name if the contract is to be put in a sub-domain:

```
service.domain.name ::= 'domain' ':' domain.name ':'
domain.name ::= qualified.domain.name | 'public' | 'private'
qualified.domain.name ::= 'valid domain name'
```

A domain name cannot be arbitrarily chosen: as service domains are intended to be used by a lot of organizations, there must be some well-defined process to choose names and make them known to others. This process will probably be carried out by a 'LINC Community Process' analogous to the Java or Jini Community Processes [JAV, JIN]. The only usable names are those defined by the *community*: it is an error to use domain names that are not granted for one's use. The special domain referred to by *public* is reserved for generic services that do not belong to any particular domain. Such public services are also subject to the LINC Community Process naming conventions. The special domain referred to by the keyword *private* is meant to be used as a kind of 'sandbox' domain. This domain cannot be exported to other locations, so the service names in this domain do not need to be approved by the community process.

**Rationale**

The system of domain names was inspired by the Java/Jini package naming process because the goals are similar: re-use of globally available software libraries. Even though there is nothing like a LINC Community Process yet, the naming process has to make sure that domain and service names are used in a way that prevents name conflicts and advocates consistent use of service names.

Domain names correspond to directory names on the user's computer or network. The directory structure has to reflect the name space specified by the LINC user's community. Many users at widely different locations may all have directories for the same domains: the Logistic Contract Space consists of the merge of all these directories. When compiling a contract, the LINC compiler will put the generated classes under the directory that is implied by the domain name that was specified in the contract. If that directory does not
yet exist, it will be created by the compiler. As the generated classes contain references to
the Java package names implied by their locations, it is an error to move these classes to a
different directory.

The service name

The second part of the service description is the service name. The service name identifies
the service defined by this contract so it can be referred to by potential users (human
‘end-users’ or other contracts searching for sub-contracts). A contract specifies exactly one
service, whose name must be given following the keyword service. The service is denoted
by a simple name:

```
service.service.name ::= 'service' ':' service.name ';'
service.name ::= simple.service.name
simple.service.name ::= 'valid service name'
```

Like the domain name, the service name implies a directory on the user’s computer. This
directory is located directly under the directory that corresponds to the domain name. If
the directory does not yet exist, it will be created by the compiler. It has not yet been
decided if service names can be freely chosen. As the generated classes contain references
to the Java package names implied by their locations, it is an error to move these classes
to a different directory or to rename them.

A service description example

The following example is derived from the contracts given in several of the references. To
improve readability, full names have been substituted for the acronyms, though:

```
description
  domain  : FreightUnitTransport;
  service  : putOnTransport;
```

5.7 The Service Actors

The service actors section specifies the types of logistic actor (i.e. provider and client)
involved in the contract. When a contract is instantiated, a specific actor instance must be
selected for each actor mentioned in this section. The syntax of the service actors section
is as follows:

```
service.actors ::= 'actors' service.provider service.client
service.provider ::= 'provider' ':' provider.type.name ';'
service.client ::= 'client' ':' client.type.name ';'
```
5.7. The Service Actors

```
provider_type_name ::= 'existing actor type name'
client_type_name ::= 'existing actor type name'
```

The service actors section names the service provider and the service client of the contract, i.e., the logistic actors that are going to provide c.q. use the service described by the contract. A 2-party contract specifies exactly one provider and one client; an actor type must be specified for each of them. Both the provider type and the client type are denoted by (qualified) type names, that may or may not yet be defined in the specified domain:

- An actor type is defined if it refers to a Java source or class file that belongs to the package corresponding to the domain and service in the Logistic Contract Space, and if the referred class file represents a Java interface that extends an (anonymous) interface specifying the current contract's provider tasks. Note that the latter interface may extend other (anonymous) interfaces as well, specifying tasks belonging to contracts other than the current one!

- An actor type is undefined if there is no corresponding Java source or class file that belongs to the package corresponding to the domain and service in the Logistic Contract Space. In this case new Java source and class files for the provider are created on the fly, extending a single (anonymous) interface specifying the current contract's provider tasks. Future releases of the LINC platform will probably prevent users from adding provider interfaces to any other domain than the special private domain.

- An actor type cannot be defined (not without expert interaction at least) if it refers to a Java class file that is part of the package corresponding to the domain and service in the Logistic Contract Space, if the referred class file represents a Java interface that does not extend an (anonymous) interface specifying the current contract's provider tasks. In this case, either another provider type must be specified, or the existing provider type must be updated to include the new extended interface. Future releases of the LINC platform will probably prevent users from changing provider interfaces in any other domain than the special private domain.

Note that it is always possible to (re)define actor types in the special private domain.

**Rationale**

The provider and client interfaces generated from service contracts represent very important information extracted from those contracts. As with any other Java interface, it is very hard to ever change them once they are published: just about anybody can have written classes implementing the interface. For this reason, a LINC community process is required to define and publish interfaces, just like the community process for Java and Jini [JAV, JIN]. This is the main reason to prevent arbitrary users from changing an existing interface (1st point above) or putting new interfaces in the Logistic Contract Space (2nd point above).
Even worse than adding a completely new interface would be to change an existing Java interface by adding new method definitions! This would force all existing providers to extend their services and provide the new service as well (and quickly, as they are invalidated right away). Surely something that would make them feel grateful. So maintaining and extending existing published interfaces is at least as important as publishing them in the first place (3rd point above).

It follows from the above that defining new actor types is not something that should be taken lightly. It should not be done frequently or after less than extreme consideration. Once defined, actors are there to stay. This may sound like a disappointment for those who believe in agile systems, but it is a reality in the world of Object Oriented software.

There is one logistic actor type that is predefined in the public domain of the Logistic Contract Space: LincBasicActor. This logistic actor type does not implement any tasks, and can be used to specify a default service provider. Any other provider has to inherit (directly or indirectly) from LincBasicActor.

**A service actors example**

The following example is derived from the contracts given in several of the references. To improve readability, full names have been substituted for the acronyms, though:

```plaintext
actors
  provider    : FreightUnitTransport.RegionalServiceCenter;
  client      : LincBasicActor;
```

The client in this example is specified as LincBasicActor, which is the basic actor type available from the Logistic Contract Space. Clients defined like this are not able to execute any tasks on behalf of the provider.

### 5.8 The Service Objects

The service objects section is used to pass state information to a contract. Service objects represent the entities that are physically handled during the lifetime of the contract, and usually keep track of the state of these entities. They can be viewed as a kind of database containing all relevant information about the object(s) handled by the contract. The LINC language provides ways to refer to, to validate, and to update this database from within a service contract. The syntax of the service objects section is as follows:

```plaintext
service.objects ::= 'objects' ( object.declaration + | 'none' ; |
```

There can be zero or more objects affected by the service specified by the contract. Each affected object is described by a name/type association, possibly followed by a default value:
5.8. **The Service Objects**

object.declaration ::= object.name ‘:\’ object.type | default.object.value | ‘;’
object.name ::= simple.object.name
simple.object.name ::= ‘unique name’
object.type ::= simple.type | map.type
default.object.value ::= ‘:=’ "null"

Object names must be unique within the scope of the current contract. Object types must be defined in the Logistic Contract Space (section 5.4); the corresponding classes must implement the **Serializable** interface². Service objects or components thereof may be used as arguments in task invocation messages (see section 5.13.1), as arguments to subcontracts services (section 5.13.2), or in assertions (section 5.13.10) and/or updates (section 5.13.11). Service objects can also be used in the pre- and postconditions of task signatures (section 5.12.1).

Service objects must be initialized when the contract is instantiated, unless a default value was specified for them. Currently, only one default value can be specified (i.e., null), meaning that the service object is left unbound. Initial values are provided by the party that instantiates the contract. Initialization upon contract instantiation is the only option for service objects for which the contract does not specify a default value. If a default value is specified for an object, the instantiating party may or may not override the default with a different value. Most contracts will require one of the parties to replace unbound service object values by something more meaningful during the course of the protocol, typically during one of the first steps. The mechanism is identical to the binding of unbound service variables (see section 5.11). If a bound service object is used in a task invocation message, the actual value in the message must be equal to the value it is bound to. If not, the task invocation message will be rejected and a **ProtocolViolation** exception will be thrown. Equality is checked by means of the equals() method appropriate for the type of the service object. Service objects can also be used in the task’s pre- and postconditions, in which case they refer to the object values that are current at the time the task is invoked. If a pre- or postcondition refers to a service object that is not an argument to the task at hand, the object must be bound when the task invocation message occurs. If it is still unbound, the task invocation message will be rejected and a **ProtocolViolation** exception will be thrown as well. Note that this can only happen for service objects that have a default null initialization. At any time, service objects defined as a map type refer to those collections of index values (or index value sequences) for which item values exist in the map, together with those item values themselves. If no item value is present in a map for a given combination of index values, that map is said to be unbound for those index values. The map can become bound for those index values just like ordinary service objects (i.e., those defined by a simple type) can become bound, i.e., by referring to them in a task invocation message.

Service objects are read-only once they are bound. Since Java does not provide language support for **immutable objects**², there is no way to prevent a contract party from trying to modify a service object. The best the compiler can do is to mark task arguments as final in

---
²If a service object corresponds to a remote object (quite likely, since service objects are meant to record shared state information), a serializable proxy object must be used in the contract.
the method representing the task, and to use *defensive copying* whenever a service object is used as an argument in a task invocation message. Though not preventing updates, any changes that are made within the offending party’s task will not disturb the other party or the protocol verifier.

**Rationale**

Given that the service objects represent important state information, it would be too dangerous (and contrary to the purpose of logistic contracts!) to allow the contract parties to freely manipulate service objects. Ideally, the language should allow for the specification of *access rights* for every service object; in the absence of a mechanism like this, access is defined as follows: contract parties have *read access*; no party has *write access*, though, and any updates to the values of service objects must take place at the *contract level*, in so-called *update blocks* (see section 5.13.11). Similarly, consistency of service object values can be checked in so-called *assert blocks* (see section 5.13.10).

**A service objects example**

Most contracts have a single service object, so the next example is slightly hypothetical. It shows two objects, one corresponding to an individual parcel and one corresponding to a set of parcels:

```
objects
  theClientParcel : FreightUnitTransport.ClientParcel;
  theClientParcelSet : FreightUnitTransport.ClientParcelSet := null;
```

The null initialization of the second service object allows the contract to be instantiated even when the set of parcels is not yet known. In this case, the set can be passed at a later time. If the object is initialized with an actual value, any value passed at a later time must equal the initial value, i.e., the service object behaves just like a pre-initialized temporary value (see section 5.11).

**5.9 The Service Parameters**

The service parameters section declares a set of parameters that are used to pass information into the contract. Parameters are a bit like service objects (except that they don’t correspond to entities being handled by the contract), and a bit like service variables (except that they are initialized before the contract is started and are passed to the newly created contract instance). The syntax of the service parameters section is as follows:

```
service.parameters ::= 'parameters' ( parameter.declaration + | 'none' ';' )
```
5.9. The Service Parameters

There can be zero or more parameters to be passed to the contract. Each parameter is described by a name/type association, possibly followed by a default initialization:

\[
\begin{align*}
\text{parameter declaration} & \quad ::= \quad \text{parameter name} \ ' : ' \ \text{parameter type} \ [ \ \text{default parameter value} \ ] \ ' ; '
\text{parameter name} & \quad ::= \quad \text{simple parameter name}
\text{simple parameter name} & \quad ::= \quad ' \text{unique name}'
\text{parameter type} & \quad ::= \quad \text{simple type} \ | \ \text{map type}
\text{default parameter value} & \quad ::= \quad ' := ' \ ' \text{null}'
\end{align*}
\]

Parameter names must be unique within the scope of the current contract. Parameter types must be defined in the Logistic Contract Space (section 5.4); the corresponding classes must implement the `Serializable` interface\(^3\). Service parameters or components thereof may be used as arguments in task invocation messages (see section 5.13.1), as arguments to subcontracted services (section 5.13.2), or as arguments in assertions (section 5.13.10) or updates (section 5.13.11). Service parameters can also be used in the pre- and postconditions of task signatures (section 5.12.1). Service parameters are unique in that they can be used in the constraints of a multiple interaction pattern (section 5.13.9) or a timed interaction pattern (section 5.13.6). In contrast to service objects, service parameters can not be used as the target of an assertion or update (see sections 5.13.10 and 5.13.11).

Service parameters must initialized when the contract is instantiated, unless a default value was specified for them. Currently, only one default value can be specified (i.e., `null`), meaning that the service parameter is left unbound. Initial values are provided by the party that instantiates the contract. Initialization upon contract instantiation is the only option for service parameter for which the contract does not specify a default value. If a default value is specified for a parameter, the instantiating party may or may not override the default with a different value. Most contracts will require one of the parties to replace unbound service parameter values by something more meaningful during the course of the protocol, typically during one of the first steps. The mechanism is identical to the binding of unbound service variables (see section 5.11). If a bound service parameter is used in a task invocation message, the actual value in the message must be equal to the value it is bound to. If not, the task invocation message will be rejected and a `ProtocolViolation` exception will be thrown. Equality is checked by means of the `equals()` method appropriate for the type of the service parameter. Service parameters can also be used in the task’s pre- and postconditions, in which case they refer to the parameter values that are current at the time the task is invoked. If a pre- or postcondition refers to a service parameter that is not an argument to the task at hand, the parameter must be bound when the task invocation message occurs. If it is still unbound, the task invocation message will be rejected and a `ProtocolViolation` exception will be thrown as well. A service parameter must also be bound when protocol flow reaches a multiple that uses the parameter in its constraints (see section 5.13.9). Note that this can only happen for service parameters that have a default `null` initialization. At any time, service parameters defined as a map type refer to those

\[^3\text{If a service parameter corresponds to a remote object (not very likely), a serializable proxy object must be used in the contract.}\]
collections of index values (or index value sequences) for which item values exist in the
map, together with those item values themselves. If no item value is present in a map for a
given combination of index values, that map is said to be unbound for those index values.
The map can become bound for those index values just like ordinary service parameters
(i.e., those defined by a simple type) can become bound, i.e., by referring to them in a task
invocation message.

Service parameters are read-only once they are bound. Since Java does not provide language
support for immutable objects*, there is no way to prevent a contract party from trying to
modify a service parameter. The best the compiler can do is to mark task arguments as
final in the method representing the task, and to use defensive copying whenever a service
parameter is used as an argument in a task invocation message. Though not preventing
updates, any changes that are made within the offending party's task will not disturb the
other party or the protocol verifier.

Rationale

The main use of service parameters is to pass contextual information and constraints.
As such, they are used mostly to constrain multiples or to constrain subcontractors of
the current contract. Since changing a constraint from within the constrained construct
can only lead to chaos, it is forbidden to set any parameter from within a multiple or
timed construct. This restriction could obviously be lessened by making the compiler
analyze the constraints in more detail; since setting parameters is something that should
take place as a first step, this refinement was not considered worth the extra effort.

A service parameters example

The following example shows a parameter that specifies the number of parcels that is to be
handled by the contract. If a contract were to handle a fixed number of items, this would
be the appropriate way to specify the number (as opposed to passing it in a task invocation
message):

```java
parameters
theNumberOfParcels : Integer;
theSubcontractor : FreightUnitTransport.Transporter;
```

The second parameter is quite interesting: it shows how one could pass a logistic actor as
a parameter to the current contract. This parameter could then be used to instantiate a
new sub-contract (see section 5.13.2). Since the actor is passed as a parameter and not as
an actor, it cannot play an active role in the current contract. If this is what is needed, the
extra actor would have to be passed as an actor in a multi-party contract (see chapter 5.16).

Rationale

One might ask why a distinction is made between logistic actors that are passed as
service actors and those that are passed as service parameters. The truth is that there
is no technical distinction; in principle either one of them could be used as an actor in a task invocation message. The decision to distinguish them nonetheless is based upon the legal status of a service contract: by registering itself as an actor in a service contract, that actor accepts all legal implications of that decision. The parameters of a service contract do not have such a legal status: from the view of the contract, they are just data to be passed around.

5.10 The Service Indexes

Service indexes are meant to pass index values (see section 5.13.9) to a sub-contract that is instantiated within a multiple. The syntax and semantics of the corresponding section is largely equivalent to the parameters section. The indexes specified in this section have to be passed in all task invocations in the current contract. As we intend to get rid of the index section of a service contract, it will not be described any further.

5.11 The Service Variables

The service variables section declares temporary variables used to model the real world entities that play a role in the execution of the service. This section starts with the keyword 'variables', which is followed by a sequence of variable declarations:

\[
\text{service.variables ::= 'variables' ( variable.declaration + | 'none' ';')}
\]

There can be zero or more variables declared in the variables section. Each variable declaration introduces a name and a type for the temporary variable:

\[
\text{variable.declaration ::= variable.name ';'} \text{ variable.type ';'}
\]
\[
\text{variable.name ::= simple.variable.name}
\]
\[
\text{simple.variable.name ::= 'unique name'}
\]
\[
\text{variable.type ::= simple.type | map.type}
\]

Variable names must be unique within the scope of the current contract. Variable types must be defined in the Logistic Contract Space (section 5.4); the corresponding classes must implement the Serializable interface. Temporary variables or components thereof may be used as arguments in task invocation messages (see section 5.13.1), as arguments to subcontracted services (section 5.13.2), or as an argument in assertions (section 5.13.10) or updates (section 5.13.11). Temporary variables can also be used in the pre- and postconditions of task signatures (section 5.12.1). In contrast to service objects, temporary variables can not be used as the target of an assertion or update (see sections 5.13.10 and 5.13.11).

Temporary variables will be created when execution of the protocol begins; in contrast to service objects and service parameters, no default initialization is allowed. Variables
behave like logical variables* in e.g. the Prolog language [CM81]. This means that they are initially unbound, i.e., that they do not refer to a value. When used in a task invocation message in the protocol section, an unbound variable will be bound to the value appearing in the actual task invocation. From that moment on, the variable refers to (is bound to) that value. If a bound variable is used in a task invocation message, the actual value in the message must be equal to the value it is bound to. If not, the task invocation message will be rejected and a ProtocolViolation exception will be thrown. Equality is checked by means of the equals() method appropriate for the type of the service parameter. Service variables can also be used in the task’s pre- and postconditions, in which case they refer to the values that are current at the time the task is invoked. If a pre- or postcondition refers to a service variable that is not an argument to the task at hand, the variable must be bound when the task invocation message occurs. If it is still unbound, the task invocation message will be rejected and a ProtocolViolation exception will be thrown as well. A service variable must also be bound when protocol flow reaches a multiple that uses the variable in its constraints (see section 5.13.9). At any time, service variables defined as a map type refer to those collections of index values (or index value sequences) for which item values exist in the map, together with those item values themselves. If no item value is present in a map for a given combination of index values, that map is said to be unbound for those index values. The map can become bound for those index values just like ordinary variables can become bound. i.e., by referring to them in a task invocation message.

Service variables are read-only once they are bound. Since Java does not provide language support for immutable objects*, there is no way to prevent a contract party from trying to modify a service variable. The best the compiler can do is to mark task arguments as final in the method representing the task, and to use defensive copying* whenever a service variable is used as an argument in a task invocation message. Though not preventing updates, any changes that are made within the offending party’s task will not disturb the other party or the protocol verifier.

Rationale

Defining temporary values as Prolog-style logical variables provides a basis for checking the values that are passed in subsequent messages between the client and the provider. It allows one to specify that the same value must be passed for a certain argument in a series of related messages. This feature is essential within a multiple construct, where a common argument value is needed to group messages together (see section 5.13.9).

Variables do not have default values since they will never be initialized upon contract instantiation. As a result, there is never any need to specify that an initial value is not yet known.

Temporary variables are often maps: almost every occurrence of a variable inside a multiple will be a map variable. This usage is so common that the compiler will even produce a warning if a non-map variable is used as a task argument within a multiple. No such warnings will be produced for service objects and service parameters used as arguments.
5.12. The Service Tasks

A service variables example

The following example shows a few simple variable declarations:

```
variables
  clientParcel : FreightUnitTransport.Parcel;
  clientParcels : FreightUnitTransport.ParcelSet;
  theParcelState : FreightUnitTransport.ParcelState;
  theFlight : map [PalletTag] of FlightNumber;
  thePallet : map [PalletTag] of FreightUnit;
```

The last two lines define variables of a map type; the maps are indexed by a value of type PalletTag.

5.12 The Service Tasks

The provider tasks section of the contract lists all tasks that the provider can execute without interaction with the client. If there are any tasks the client should execute on behalf of the provider, they can be specified in the client tasks section.

```
service_tasks ::= provider_tasks [ client_tasks ]
provider_tasks ::= "provider" "tasks" task_signatures
client_tasks ::= "client" "tasks" task_signatures
task_signatures ::= ( task_signature ; ) + | "none" ;
```

All tasks that are part of the contract must be specified by means of task signatures, which are described below. If a contract party does not need to perform any tasks on behalf of the other party, the keyword none should be specified instead of the list of task signatures. Alternatively, if the client does not have to perform any tasks on behalf of the provider, the client tasks section can be omitted altogether. The provider tasks section must always be present.

Rationale

A service contract will seldom be specified as a monolithic job which is passed from the client to the provider, then executed by the latter, after which the provider informs the client of the completion of the job. Instead, most jobs will be executed in a number of separate tasks, where these tasks are assumed to be the largest 'chunks' of the job that can be executed by the provider without interaction with the client. So while the job as a whole is not monolithic, each task is. This means that the client does not have to know anything about what goes on inside a task; it only has to know its name, and maybe provide some data the provider needs to execute the task. These data are given as arguments to the task. A task may specify data to be journalled by means of a journal value (see section 5.14). Nothing is said about the order in which the tasks
will be executed; this information must be specified in the service protocol, which is
described in section 5.13.

The client also has to know if the provider might encounter situations it cannot handle
during the task. In such cases the provider might fail to terminate the task successfully,
and generate an exception instead. In order to keep a contract clear, it is advisable
to keep the handling of such exceptional conditions separate from the normal protocol
flow. This is done by allowing a task to throw an exception. Any exceptions that might
occur should be specified in a task's definition, though, in order to prepare the client
for the worst and allow it to handle the situation properly. It is important that a client
can find out all required details about the state of the aborted process, so it can resume
the protocol after it has handled the exceptional situation. The only way this can be
accomplished, is by the provider putting enough information in the exception for the
client to analyze the situation.

In the preceding explanation it was silently assumed that it is always the provider who
executes a task on behalf of the client. This is only part of the story though, as there
may also be circumstances in which the provider wants the client to execute some task.
So in reality the situation is purely symmetrical, as both parties can ask the other party
to execute some task.

5.12.1 Task signatures

A task specification or task signature defines the arguments of the task. It specifies whether
or not the task generates some data for the journal (see section 5.14), whether there are
pre- or postconditions on the arguments and/or the journal value, and whether there are
any exceptions that might be thrown. A task signature is defined as follows:

\[
\text{task_signature} ::= \text{task.name} \\
\text{task.arguments} \\
\text{task.precondition} \\
\text{task.postcondition} \\
\text{task.exceptions} \\
\text{task.journal.value}
\]

Task name

The task name is a simple name:

\[
\text{task.name ::= simple.task.name} \\
\text{simple.task.name ::= 'valid task name'}
\]

There may be more than one task with the same name, provided that the argument types
of each task are different. If more than one task has the same name, the name is said to
be overloaded.
5.12. The Service Tasks

Rationale

The decision to allow task name overloading within a party signature was taken despite many people's conviction that overloading of method names is (potentially) confusing and should be avoided [Blo01, Mey97]. The main reason to allow for it is that it is allowed in Java and thus may be needed if one needs to write a LINC contract for existing Java objects.

Task arguments

Most tasks will require data to be able to execute. Such data can be passed by means of the task's arguments:

\[
\begin{align*}
\text{task.arguments} & ::= \text{`(`} \ [ \text{task.argument.list} \ ] \text{`)'} \\
\text{task.argument.list} & ::= \text{task.argument} \ (\text{`,'} \text{task.argument} \ ) \ * \\
\text{task.argument} & ::= \text{[} \text{task.argument.name} \text{`} \text{'} \text{task.argument.type} \\
\text{task.argument.name} & ::= \text{simple.argument.name} \\
\text{simple.argument.name} & ::= \text{`unique argument name'} \\
\text{task.argument.type} & ::= \text{name}
\end{align*}
\]

The minimum that must be specified for each argument is the argument's type. The argument type names must refer to existing types defined in the Logistic Contract Space. These types must implement Serializable. If there are pre- or postconditions (see below) that need to refer to argument values, it is necessary to specify an argument name too; this name can then be used in the conditions. Argument names must be unique within the argument list of a task.

The actual objects passed as task arguments have to be valid with respect to the precondition specified for the task. Within the receiving party's task, their value must never be changed. Unfortunately, this restriction cannot be enforced by the compiler; the best it can do is designate task arguments final in the generated Java code and use defensive copying.

Rationale

As the LINC framework is designed on top of Java RMI, all LINC actors are implemented as Remote objects. In principle, all object instances to be passed between remote actors can be either Serializable or Remote; the current framework requires them to be Serializable though. Since the instances are serialized before they are sent and will be de-serialized at the receiver's end, the latter effectively holds a copy of the data originally sent, not the data itself. Changing such a copy would cause the objects at both ends to become inconsistent with each other. Note that even if a remote reference were passed, it would be impossible for a party to change the value of the object and use it in subsequent task invocation messages, at least not if the same variable is involved. This is because of the semantics of logical variables, which would cause an exception if the values of a task argument before and after the invocation would differ.
All this means that task arguments do not provide a means to pass information back to the calling party. Any information that the provider wants to make available to the client has to be communicated by means of the *service journal* (which is described in section 5.14).

**Task pre- and postconditions**

The task signature may contain a *precondition* to put restrictions on the values allowed as arguments. There might be restrictions on single argument values that cannot be expressed by means of the argument types (e.g., a restriction might be that a certain argument value is *odd* or *even*). More often though, it might be necessary to check a certain relation that must hold between the arguments or between arguments and some service parameters and/or variables. The signature can also contain a *postcondition* which specifies a condition that must hold when the task terminates. The pre- and postconditions can be specified as follows:

\[
\begin{align*}
\text{task.precondition} & := [\ '\text{pre}' \ \text{boolean.expression} ] \\
\text{task.postcondition} & := [\ '\text{post}' \ \text{boolean.expression} ]
\end{align*}
\]

The syntax of boolean expressions is defined to be equal to the syntax of boolean expressions in Java\(^4\). The expressions may refer to the arguments of the task at hand, to service objects (section 5.8), service parameters (section 5.9), and service variables (section 5.11). Service objects, parameters, variables, and components thereof must be *bound* at the time the pre- or postcondition is evaluated (i.e., at the beginning and end of the task’s execution). Additionally, a postcondition may refer to the task’s journal value by means of the task name. The expressions may contain method invocations and/or primitive operations, provided these methods and operations are defined for their argument types\(^5\).

If a precondition is present, this means that correct behaviour of the task is only defined when the precondition evaluates to *true*. If it happens to evaluate to *false*, a *PreconditionViolation* exception will be thrown. Preconditions will be checked just before the task message is actually sent; if a *PreconditionViolation* is thrown, the exception must be handled internally by the sender’s business logic. To the receiving party, it appears as if the offending task invocation message has not (yet) been sent. The sender can retry to invoke the task until it is successful. Note that the *PreconditionViolation* exception must not be listed with the task’s exceptions, as it cannot be handled at the protocol level.

\textit{Rationale}

Preconditions are meant to protect actors from ignorant or malicious clients; they validate the input data itself as well as the relation between this data and the current

\(^4\)To maintain stylistic conformity with the rest of LinC, the Java operators \&\&, ||, and ! may also be written as \texttt{and}, \texttt{or}, and \texttt{not}, respectively. In our examples, we will consistently use the latter forms.

\(^5\)In the current version of the LinC compiler expressions are only checked with respect to their syntax. Semantic validation of the expressions is deferred to the Java back-end compiler.
5.12. The Service Tasks

state of the protocol. Together with postconditions, they are an integral part of the ‘design by contract’ paradigm, and without them, LINC specifications would not even come close to being contracts. The extra safety offered by pre- and postconditions is especially relevant in the LINC context, where the actor’s business logic might be maintained by parties that are operating independent of each other.

If a postcondition is specified, this means that the condition is guaranteed to be met when the task returns. The business logic of the task is obliged to check the postcondition by means of a special validateReturn () call\(^6\); if validateReturn () succeeds, the task is allowed to return. If the postcondition evaluates to false, validateReturn () will throw the PostconditionViolation exception, to be handled internally to the business logic of the party that unsuccessfully executed the task. To the sending party, it appears as if the task has not ended yet. The executing party can retry until the postcondition test is passed, and then return. Note that the protocol verifier double-checks the condition; if the executing party should try to skip the postcondition check, the protocol verifier will notice and throw a ProcessError exception to the original sending party; the same exception will be thrown if the executing party ignored the PostconditionViolation exception and returns nonetheless. In either case, all the client knows is that something is seriously wrong at the provider’s side. Thus, even though the original sender will never see any PostconditionViolation, it must be prepared to handle ProcessErrors. Note that the ProcessError exception need not be listed with the task’s exceptions, as it cannot be handled at the protocol level.

**Rationale**

The handling of postconditions may appear overly complex. Maybe it is, but it is the best one can do without changing the Java Virtual Machine (JVM). The problem is that postconditions should be validated upon method return. In fact, their testing should be integrated with the return, and it should be the return statement itself that throws the PostconditionViolation! If the LINC platform gains wider acceptance, changing the JVM might be the best way to go. If the Java language itself would ever be extended with pre- and postconditions, it might be possible to use that mechanism. Note that the ProcessError exception has not been invented just for the purpose of double-checking returning parties: it is a general error that can be thrown by any party that cannot meet the terms of the contract any longer. It is used in cases that are so far from normal operating conditions that they cannot be foreseen or handled within the contract (see also section 5.12.1).

Ideally, expressions used in pre- and postconditions do not throw exceptions themselves. However, since these expressions may use operations that are defined outside the contract, this is impossible to guarantee. In order to ensure consistent behaviour even in case of an exception thrown during expression evaluation, any such exception is defined to cause

---

\(^6\) Actually, every return from a task must be validated, no matter if a postcondition is present. The code stubs generated by the LINC compiler will include this call; if extra returns are added to the implementation of a task, extra validates have to be added too.
that expression to return the value \texttt{false}. As a result, such exceptions will give rise to a subsequent \texttt{PreconditionViolation} or \texttt{PostconditionViolation}. The latter exceptions will be treated in the usual way.

\textbf{Task exceptions}

Any user-defined exceptions that can be thrown from within a task are defined as follows:

\begin{verbatim}
task.exceptions ::= [ `throws` task.exception_list ]
task.exception_list ::= task.exception (` `, task.exception ) *
task.exception ::= name
\end{verbatim}

Exception names must refer to existing exception types in the Logistic Contract Space. Information about the execution status of the task that ended with an exception may be retrieved by inspecting the exception. Exceptions must be handled in the usual way by the business logic of the party that invoked the task. In many cases, the exception will change the communication flow between the contract parties. For that reason, exceptions must also be accounted for in the protocol; the \texttt{try construct} is used for this. Exceptions like these are classified as \textit{contractual exceptions*}, since they are defined and handled within the contract. See section 5.13.5 for details.

Apart from user-defined exceptions, any task can throw a \texttt{ProcessError} exception or a \texttt{TimeoutError}. The former exception may be thrown if the party that is executing the task finds itself in a situation that cannot be recovered from (bankruptcy might come to one’s mind). The latter exception might be thrown if a task is invoked after such a long period that the party cannot be expected to honor the request anymore. The exact length of the period is not specified; it is up to the receiving party to decide if it has been patient enough. Neither of these exceptions must be explicitly mentioned in the task exception list; they can be handled in a try construct though. Both exceptions are examples of \textit{general exceptions*} that will probably be handled by \textit{informal handlers*}. See section 5.13.5 for more details.

\textbf{Rationale}

The existence of special exceptions makes it clear that there may be cases that are outside the contract’s control. Contract writers can and should use regular exceptions for all special cases that can be foreseen, and for cases whose handling can be described in an exception handler within the protocol. If cases that cannot be described, the special exceptions can be used to escape to an exception handler that resorts to communication that is not specified within the contract.

\textbf{Task journal values}

Tasks can be specified to include a result value in the \textit{journal tree*} when they terminate. This is done by including a \textit{journal type} in the task signature:
5.12. The Service Tasks

\[ \text{task.
s.
\text{journal.type} ::= [ \text{'journal'} \text{name} ]} \]

The journal type name must refer to an existing type defined in the Logistic Contract Space. This type must implement \text{Serializable}.

\textit{Rationale}

As was said above, tasks may have arguments but no return value. As the task arguments are read-only, tasks cannot use them to pass information back to the party that sent the task invocation message. Thus, the only way to make results known to a party invoking a task is by means of the journal value! Contract parties will be notified of changes in the journal tree, after which they have (read-only) access to all information in the tree, including the journal values.

The reason to forbid tasks to have a regular return value (like functions) is that such return values are incompatible with the interpretation of the protocol (see 5.13): the protocol is specified as a \textit{play}, in which the actors wait for their turn to make some utterance. Utterances are one-way: if a reply is expected, the actor who made the utterance just waits for another actor to generate a response.

The 'quit' signature

There is one task signature that is quite special: the \textit{quit} signature. This signature defines a special task that must be implemented by a party if its opponent may end a multiple interaction pattern by sending a \textit{quit message} (see section 5.13.9). The \texttt{quit ()} task has one argument, which is the keyword \texttt{exit}. \texttt{quit()} cannot have a pre- or postcondition, it cannot throw any exceptions, and it cannot have a journal value. An example is given below.

\textit{Rationale}

The \texttt{quit ()} task must be included in a party's task signature to make it clear that the party has to be prepared to accept this message from another party that can force termination of a multiple construct. Indeed, the receiving actor has to implement the \texttt{quit ()} task in a way that is consistent with the protocol. So even though it would in principle be possible to omit the \texttt{quit ()} tasks from the party signatures and infer them from the protocol, the proposed syntax was considered more explicit and therefore preferred.

5.12.2 Some task signature examples

The following example shows some provider tasks. Client tasks follow exactly the same pattern:
provider tasks
charge (theParcelSet : FreightUnitTransport.ParcelSet)
  pre not theParcelSet.isEmpty ()
  throws FreightUnitTransport.SetTooLarge;
  discharge (ParcelSet)
  journal InvoiceData;
  announce (theParcel : Parcel)
  pre theParcelSet.contains (theParcel) and not theParcel.isAnnounced ()
  throws ParcelTooLarge;
  enter (theParcel : Parcel)
  pre theParcel.isAnnounced ()
  journal ParcelIdentity;
  clear (theParcel : Parcel)
  pre theParcel.isAnnounced ()
  journal ParcelIdentity;
  deliverAtDestination (theParcel : Parcel)
  pre theParcel.set.entered ()
  throws ParcelDamaged, ParcelLost
  journal ParcelIdentity;
  send_16pack (s : ParcelSet)
  pre s.extent () mod 16 == 0;
  quit (exit);

In the example, charge () is supposed to charge the provider with handling a given set of parcels. The set must not be empty, and the exception SetTooLarge may be thrown when the actual set of parcels passed to the charge () operation is larger than the provider can handle. Note that we used qualified type names here; in the remaining tasks in this example we will assume that the types are defined in the same domain and use simple names instead. Announce () is used to announce parcels for handling; it is a violation of the task’s precondition to pass a parcel that is not in the set passed to charge () or one that has already been announced. Both enter () and clear () assume that the parcel given to them has already been announced. They will both put a value of type ParcelIdentity into the journal; this value might be used for tracking. The task deliverAtDestination () is another example of a task that might end with an exception. It supposedly defines the delivery of some parcel at some client’s location. But that might prove impossible if the parcel gets lost on the way (which never can happen, as we all know). If it still happens, the task will throw the exception ParcelLost. If the parcel gets damaged on the way, the exception ParcelDamaged will be thrown. In either case, the client might handle the exception by trying to find out if there was an insurance policy covering the parcel’s transport or the parcel’s contents itself. Send_16pack () shows a (admittedly hypothetical) task whose argument is a set of parcels with the condition that the number of parcels in the set must be a multiple of 16. It is assumed that values of type ParcelSet support the extent () operation, and that the result of this operation is an integer value. If not, the back-end Java compiler will generate error messages. The last line provides an example of the special quit () signature, which is required if the opposite party may explicitly end a multiple construct (see section 5.13.9).
Rationale

Several methods in this example have a precondition that states the assumptions about the state of execution. E.g., it is explicitly mentioned in \texttt{enter} ()'s precondition that the given parcel was announced before. In real contracts, such state-related preconditions are seldom necessary, since the order of execution is stated in another way: by the protocol itself.

5.13 The Service Protocol

The service protocol specifies the allowed interactions between the actors of the contract. It does so by describing the order in which the parties may take specified actions, i.e., by defining a protocol to which both parties must adhere while executing the contract. The protocol is \textit{symmetrical}, i.e., there is no language-defined distinction between the various logistic actors' roles.

Rationale

Symmetry is required in order to be able to describe arbitrary protocols. Quite often a protocol will be simple, just specifying what the provider will have to do on behalf of the client, and in what order. Sometimes more elaborate interaction may be required, e.g., when a client wants to get involved in parts of the execution of a contract. This might range from a simple \textit{handshaking} protocol, in which the client wants to be informed of completion of each individual task of the service before he grants permission to the provider to continue executing the rest of the contract, to a much more elaborate interaction in which the client has to provide information enabling the provider to continue executing the contract. This would be essential to be able to specify \textit{just-in-time} services. Also, this facility will often be used in exception processing.

During execution of the contract, all interactions between the client and the provider will be monitored and all violations of the protocol will raise a \texttt{ProtocolViolation} exception. This exception cannot be listed in the task signatures or caught in a monitored interaction pattern, and should be handled by the business logic of the party that tried to send the offending message. The protocol state will not be affected, and the intended receiver of the message will not even notice that something happened. The protocol will only continue as soon as one of the contract parties sends a message that conforms to the protocol again.

Protocol execution

Before we will discuss the details of the various protocol constructs, it is helpful to spend some time in defining what it means for contract parties to 'execute' a protocol. Basically, protocols do not change the way parties interact: they continue to interact by sending messages to each other or by calling tasks upon each other (below we'll say more about the
difference between tasks and messages). The protocol does not play an active role here: it just sits there and waits until a contract party tries to invoke a task defined by one of the other parties. No matter how quick this task will execute, its beginning and ending are distinguished as separate protocol events. These protocol events play a most important role with respect to the monitoring of the interactions: either they change the state of the protocol, or they cause a ProtocolViolation exception to be thrown. Apart from the beginning and ending of tasks, there are no other events that can change the state of the protocol.

Rationale

The fact that the beginning and ending of task invocations are the only events that can change the state of a protocol is an immediate consequence of the fact that all interactions between parties involve a task invocation and the fact that the protocol itself is passive. Note that we did exclude the beginning and ending of the protocol itself. While those events are quite important in relation to the execution of the contract as a whole (see also section 9.2), we focus here on the interactions that take place while the protocol is active.

The distinction between two different events, one corresponding to the start of a task and one to its end, is a direct consequence of the wish to be able to specify concurrency: if every task were a message, it would be meaningless to think in terms of concurrent execution. Of course it would be possible to exchange a second message at the end of the computation initiated by the first message. But doing so would not only amount to discarding a high-level construct and then re-implementing it by means of lower-level constructs, it would also complicate the interaction between concurrent constructs and messages: in particular it would necessitate the introduction of two different kinds of messages. For this reason the conventional 'method' based interpretation of tasks was deemed more natural; even more so because methods can be used to pass messages but not vice versa.

It is important to understand that the tasks to be executed by the contract parties are implemented as methods associated with the objects representing the parties. As such they might be used to implement some algorithm that will be executed by the actor. Execution of this algorithm will take a certain amount of time, so the task's end might be significantly later than its beginning. On the other hand, the interaction could well be nothing but a short message exchange. Such a message might be used to communicate an execution state, or as a signal to start some time-consuming computation by the provider's business logic. As neither the language nor its implementation enforces any time constraints, it is up to the implementor of a task to make sure that the task behaves in a way that is consistent with the protocol. This means that any postconditions of a task should be satisfied upon return, and that any journal values should be available by that time. For this reason it is highly unlikely that 'message tasks' are associated with postconditions and/or journal values. Since both kinds of tasks are initiated by means of a task invocation message, it is important to keep the distinction in mind when writing or reading protocols.
5.13. The Service Protocol

Rationale:

Of course it would have been possible to introduce a language distinction between method tasks and message tasks. Since method tasks can be used as message tasks in a straightforward way, there was no reason to extend the language.

Protocol state

When describing the various protocol constructs below, we will often refer to the beginning and ending of a construct. As will be clear from the foregoing, the beginning and ending times of a task execution are directly related to the receiving of a task invocation message and the completion of the invoked task: the interpretation of the beginning and end of task invocation messages is immediate. Since none of the other protocol constructs corresponds directly to a physical interaction and since all protocol events are generated by task invocation messages, the beginning and ending of all protocol constructs is ultimately determined by the task invocation messages that are exchanged under those constructs.

The beginning and ending of tasks determine the state of the protocol as a whole as well as the state of all constructs under which the task is executed. This state will often be referred to in the descriptions of the various protocol constructs. Every construct has a state, though not every aspect of a state applies to every kind of construct:

- Constructs are called active if tasks are currently executing under them, inactive if no task is currently executing under them and if they contain tasks that have not been started yet, or completed if no task are currently being executed under them and there are no tasks left to be executed under them. This rule applies to all protocol constructs, with the restriction that simple interaction patterns can only be active or completed.

- Furthermore, constructs are called open if they contain nested constructs that have not started yet, and closed if there are no nested constructs left to be started. This does not apply to task invocation messages.

Protocol ambiguity

It is important to note that a protocol must be written such that ambiguous situations are avoided. Ambiguity would arise when a given task invocation message, given the state of the protocol, can be executed in more than one follow-up path through the protocol. When this should happen, the new state of the protocol can not be determined, and thus

---

When we say a task is executed 'under' a construct, we mean that it is executed in a simple interaction pattern (see section 5.13.1) that is nested within one or more levels of other constructs. The term 'construct' is used loosely here, to denote any kind of interaction pattern that may contain nested interaction patterns.

While the Active/Inactive/Completed distinction applies to the entire (sub)protocol headed by some construct, the Open/Closed distinction only applies to the construct itself!
the protocol would not uniquely specify what may happen next. As some constructs may easily lead to protocol ambiguities while others will never do so, the following sections will elaborate on the issue whenever appropriate.

Rationale

The concept of ambiguity is closely related to ambiguity of grammars and languages. A grammar is said to be ambiguous if there exist sentences to which it assigns more than one parse tree; a grammar is inherently ambiguous if there exists no equivalent grammar that is unambiguous. Similar definitions can be applied to protocols. In practice, there is another classification that is more relevant to practical situations. This classification is related to the properties of parsing algorithms: for instance, programming language grammars are often required to be $LL(k)$ or $LR(k)$ (with $k = 1$, usually) [ASU85]. This $k$ is called the lookahead, and refers to the number of input symbols the parser may look at in order to decide how to interpret the current input symbol.

In the case of our protocol verifier, there are two possible ways to handle look-ahead: one way would allow parties to execute a task even though it is not known yet to which protocol path that task belongs; the other way would require parties to gather task invocation messages without immediately executing them. Neither method is feasible though: the first way is impractical since it is not known when (if ever) the ambiguity will eventually be resolved. Furthermore, this approach would complicate the protocol verification mechanism, as variables might become bound in one continuation path but not in others. Even worse than that is the problem of protocol errors that can no longer be given on the spot but may have to be delayed until it is clear which path through the protocol is being followed. Delaying protocol errors would be incompatible with the very ideas behind protocol monitoring. The same holds for the second approach: decoupling the sending of a task invocation from the execution of the task invoked renders the total idea of sequential and concurrent execution useless. Another reason why the second approach is generally not feasible is, because it would usually be impossible for a party to send a task invocation message before the previous task is completed. E.g., a party might need to read results from the journal tree before it can decide what to do next; also, a task’s arguments might depend on the results of the previous task. So in contrast to most programming language compilers, the LINC compiler cannot use look-ahead to resolve the ambiguity. As a result, every task invocation message must uniquely determine the path to be followed.

Unfortunately, it is not always possible for the compiler to determine if a protocol may lead to ambiguous situations. This means that protocol ambiguities may only be discovered during execution of the contract\(^9\). When a protocol ambiguity is detected in runtime, an AmbiguousProtocol exception will be thrown. This exception is a subclass of the ProtocolViolation exception mentioned before. It cannot be listed in task signatures or be caught in a monitored interaction pattern.

\(^9\)The initial version of the LINC compiler cannot detect any ambiguity in compile time, so all of them will only be found during runtime. In a sense, this allows for more general contracts, for not all possible ambiguities will actually occur in runtime. This means that an ambiguous protocol may be executed without any ambiguity ever occurring in practice! Had the compiler been more restrictive, it would have refused to compile the contract altogether.
5.13. The Service Protocol

The protocol section

The protocol section specifies the interactions between the contract parties. It is defined as follows:

```
service.protocol ::= 'protocol' interaction_pattern
```

The interaction pattern is defined by means of a number of constructs that specify various specific types of interaction pattern:

```
interaction_pattern ::= simple_interaction_pattern
                      subcontracted_interaction_pattern
                      selective_interaction_pattern
                      optional_interaction_pattern
                      monitored_interaction_pattern
                      timed_interaction_pattern
                      sequential_interaction_pattern
                      concurrent_interaction_pattern
                      multiple_interaction_pattern
```

The various interaction patterns will be described below. Even though the syntax allows any interaction pattern to be used at the top level of the protocol, there is one semantic restriction: the protocol as a whole is not allowed to throw any contractual exceptions (see section 5.13.5). The LiNC compiler will generate a error for any such exception that is not handled at some level in the protocol.

5.13.1 Simple interaction patterns

The simple interaction pattern specifies a single communication act between two contract parties. So (assuming a 2-party service contract) either the client performs a communication act directed towards the provider or the provider performs a communication act directed towards the client. The communication act will activate a task defined by the receiving party (i.e., in the provider tasks section or the client tasks section). In the following, we will often use less formal terminology, and say that a contract party sends a task invocation message* to the other party. Such a message is defined as follows:

```
simple_interaction_pattern ::= [ direction '=' ] task_invocation_message ';'
direction ::= client_to_provider | provider_to_client
client_to_provider ::= 'client' ['to' 'provider']
provider_to_client ::= 'provider' ['to' 'client']
task_invocation_message ::= task_name task_arguments
task_name ::= simple_task_name
simple_task_name ::= 'valid task name'
task_arguments ::= '(' [ task_argument_list ] ')
task_argument_list ::= task_argument ( ',' task_argument ) *
```
As shown, a simple interaction pattern may contain a sender which is either the client or the provider. Optionally, it may also specify the receiver, which must not be equal to the sender (i.e., a party is not allowed to send a message to itself). Usually it is the client that sends messages to the provider to activate one of the latter party's tasks, so if no sender is specified, the message is assumed to originate from the client and to be directed to the provider. The task invocation message following the sender must consist of a task name followed by the task argument list.

\[
\text{task_argument} ::= \text{service.object.reference} \\
| \text{service.parameter.reference} \\
| \text{service.variable.reference} \\
| \text{service.index.reference} \\
| \text{multiple.index.reference}
\]

\[
\text{service.object.reference} ::= \text{simple.object.name} | \text{map.indices} \\
\text{service.parameter.reference} ::= \text{simple.parameter.name} | \text{map.indices} \\
\text{service.variable.reference} ::= \text{simple.variable.name} | \text{map.indices} \\
\text{service.index.reference} ::= \text{simple.index.name} \\
\text{multiple.index.reference} ::= \text{simple.index.name} \\
\text{simple.object.name} ::= \text{'valid service object name'} \\
\text{simple.parameter.name} ::= \text{'valid service parameter name'} \\
\text{simple.variable.name} ::= \text{'valid service variable name'} \\
\text{simple.index.name} ::= \text{'valid multiple index name'} \\
\text{map.indices} ::= \text{map.index (',', map.index) *} \\
\text{map.index} ::= \text{'}0\text{' multiple.index.reference}
\]

Task arguments may be references to service objects (section 5.8), service parameters (section 5.9), service variables (section 5.11), or components thereof. They may also refer to service indexes (section 5.10) or multiple indexes (see section 5.13.9). The task invocation message must conform to one of the task signatures defined for the receiving party, i.e., the types of the values specified in the argument list must be in one-to-one correspondence to the types specified in exactly one task signature for the task name in question. If the task invocation message occurs within the scope of one or more multiple constructs (section 5.13.9), and/or if service indexes (section 5.10) were passed to the contract, some additional rules apply:

- When a task invocation message occurs inside one or more multiples, it must refer to each of the multiple indices: each multiple index corresponds to a specific task argument called an index argument. Index arguments of a task invocation message must refer to the proper multiple index by using the index name preceded by a @-sign\(^\text{10}\). It is an error for a task invocation message inside a multiple not to include a reference to each multiple index.

- Service indexes passed to the contract are treated like local multiple indexes, so these too need to correspond to index arguments.

\(^{10}\text{This requirement might be dropped in a later version of the language, making the @ optional.}
- All task arguments that are not index arguments will usually be references to service variables that are defined by map types (see section 5.4); such references must be indexed by the multiple indexes. As a result, each branch in the multiple will effectively be associated with its own instance of the service variable's item type. Service objects and parameters may also be map references, but will often be shared over the different multiple branches. See section 5.13.9 for more details.

When the opposite party receives the message, it will activate the task that is mentioned in the message. The simple interaction pattern will stay active as long as it takes for the opposite party to complete the task. As said in section 5.12.1, values passed in a task invocation message should never be changed by the receiving party. Exceptions thrown by the activated task will be returned to the party that invoked the task: with regard to the protocol, they will be be propagated to the context of the simple interaction pattern, until there is a handler for the exception (see section 5.13.5). So if a task has thrown an exception, it is as if the simple interaction pattern itself had thrown the exception. If a journal type was specified for the task, a value of this type will be included in the journal upon successful completion of the task (see also section 5.14).

Rationale

While the use of service objects, parameters, and variables as arguments in task invocation messages is straightforward, the requirements necessitated by including the latter in multiple constructs might be less obvious. The problem is that the protocol verifier needs to infer each message's logical position within the protocol structure; the enclosing multiple merely states that there will be a certain number of instances of the nested interaction pattern, creating an equal number of 'branches' in the multiple construct. When invoking a new task at the opposite party, the sender does not specify which branch this new message belongs to. Or does it? Indeed it does, by using a specific argument position as a 'branch selector'! This is why we need to have an index argument for every multiple construct that surrounds the task invocation message.

Now about the other arguments, which will be maps in most cases: using regular variables would force them to be equal in all branches (see also section 5.11). If we want them to be different, either we would need map variables or we would need local variables declared within the scope of each multiple construct. It was decided to favour map variables over local variables. Local variables will probably be added at a later stage.

Some examples

The first example shows a number different ways to invoke a provider task:

```plaintext
announce (theParcel);
client : announce (theParcel);
client to provider : announce (theParcel);
```
All task invocation messages in this example are functionally equivalent; the second form may be the most appropriate to use in a 2-party contract. The next example shows a task invocation message that is supposed to be embedded within a multiple having the index 'thePalletTag : PalletTag'.

```
client : transport (@thePalletTag, theDestination[@thePalletTag]);
```

The task involves the transport of a certain pallet to a destination that is specific for that pallet. As the task invocation message is embedded within a multiple, it must include a reference to a multiple index. In this case, the first argument refers to the multiple index, i.e., thePalletTag. The second argument is used to specify the pallet's destination. As this contract is supposed to specify that the pallets are to be sent to different destinations, a map variable must be used here. The map is indexed by the pallet tag.

### 5.13.2 Subcontracted interaction patterns

The *subcontracted interaction pattern* makes it possible to instantiate an entire service contract as a sub-contract of the current contract. Common services can thus be factored out and made into separate service contracts. The subcontract's actors, service objects, service parameters, and service indexes act as parameters that must be passed when the subcontracted service is instantiated. The syntax used to instantiate a sub-contract is given by the following rules:\(^{11}\):

```
subcontracted_interaction_pattern ::= 'new' service_name '(' service_arguments ')' ';'
service_name ::= 'existing service name'

service_arguments ::= service_provider ',' service_client
                     [ ',' service_objects ] [ ',' service_parameters ] [ ',' multiple_indexes ]

service_provider ::= 'provider' | 'client' | service_parameter_reference | service_variable_reference

service_client ::= 'provider' | 'client' | service_parameter_reference | service_variable_reference

service_objects ::= service_object_reference ( ',' service_object_reference ) *

service_parameters ::= service_parameter_reference ( ',' service_parameter_reference ) *

multiple_indexes ::= multiple_index ( ',' multiple_index ) *

multiple_index ::= service_index_reference | multiple_index_reference
```

The service name must be the name of an existing service in the Logistic Contract Space. The logistic actors may refer to the provider and the client of the current contract by using the respective keywords, or to a service parameter (section 5.9), a service variable (section 5.11), or a component thereof. The types of the actual provider and the actual client

---

\(^{11}\)These rules are meant as a specification for human readers; they are not suitable to generate a parser. The problem is that the rule for service arguments distinguishes between several kinds of arguments, while those kinds can only be known by looking at the subcontracted service contract itself.
must match the types that are specified in the subcontract's actors section. If a service variable or parameter or a component thereof is used as an actual provider or client, the declared type of that parameter, variable, or component must be a descendent of **LincBasicActor**; furthermore, each parameter, variable, or component must be **bound** to different parties when the protocol flow reaches the subcontracted interaction pattern. Additional restrictions are dependent on the context of the subcontracted interaction pattern:

- **Subcontracted interaction patterns that are outside a process block (section 5.13.12)** must have the same provider and the same client as the current contract.

- **Subcontracted interaction patterns that are inside a process block must have that block's executing party as their client.** The provider of a subcontracted interaction pattern inside a process block may be specified as any or as a service parameter or variable that is declared to be of an actor type (i.e., a descendant of **LincBasicActor**). In particular, the provider of the subcontracted interaction pattern must not be equal to the current provider.

- The keyword **any** may only be used inside process blocks. It specifies that the provider does not need to be known at the contract level and may be selected in the actor's process logic. It does **not** mean that the actual provider is irrelevant!

**Rationale**

The restriction that (outside a process block) the provider and the client of a subcontract must be the same provider and the same client as that of the enclosing contract is related to the interpretation of this kind of subcontracting as a modularization mechanism. Subcontracting outside a process is specifically not intended to include information about what other actors are doing at the same time. It is possible though to include information about the **delegation** of services to third parties. This use of subcontracting is restricted to process blocks, where the subcontract's client must be the party that executes the process block (otherwise it wouldn't be a case of delegation). Note that all the mentioned restrictions are motivated by logistic reasons; there are no technical reasons to enforce them, and they might be dropped later.

While the current syntax allows for the instantiation of a new service contract, it does not provide for the specification of interactions that are to take place under control of an existing, long-running contract. Since this situation is quite common (e.g., a contract for assembling a car from a number of sub-assemblies might need to refer to the production of those sub-assemblies, which are produced by a service that will be delegated to another provider under a long-term contract that covers many instantiations of the current, delegating, contract). See also section 11.2.6.

The service actors must be followed by actual values for the service objects (if any), the service parameters (if any), and the service indexes (if any). The references (or components thereof) passed as service objects of the sub-contract must form a subset of the service objects passed to the current contract; they must be bound when the protocol flow reaches the subcontracted interaction pattern. The references (or components thereof) passed to
service parameters of the sub-contract must form a subset of the service parameters of the current contract; they must be bound when the protocol flow reaches the subcontracted interaction pattern. If a service is to be subcontracted inside a multiple interaction pattern and/or the contract has service indexes declared in the corresponding section, it is necessary to pass the current multiple indexes to the subcontracted service.

Rationale

The latter restriction is necessary because the subcontract's protocol is substituted for the sub-contract instantiation, so the task invocation messages occurring in the nested protocol must still be assigned a branch at the higher level protocol. A service index is allowed to correspond to a service object of the subcontract. This might result in the same object being passed twice to the same subcontracted interaction pattern!

The restrictions on the use of the various other entities as arguments to sub-contract instantiations follow more or less from the restrictions on those entities. Service objects of the current contract can be passed as service objects of a subcontract, but service parameters or service variables cannot. This is because the sub-contract may try to update its service objects, something that is not possible for parameters and variables. Service objects must be bound in order to be passed; this restriction is necessary to guarantee that the same service object is used in both contracts. Service parameters of the current contract may be passed as service parameters of the subcontract, but service objects and service variables cannot. This restriction has no technical reason, and might be dropped if there is any advantage in doing so. Service parameters must also be bound in order to be passed; this restriction is necessary to enforce consistency between parameter values.

When a subcontracted service is instantiated, the interaction pattern specified by the protocol of the subcontracted service will effectively replace the subcontracted interaction pattern and become a sub-protocol of the current protocol. Any contract can be invoked as a subcontract; in particular a contract can instantiate itself (i.e., recursive contracts are allowed). The actual time of instantiation is left unspecified, but has to fulfill the requirement that the sub-contract is instantiated when a task is invoked that might be executed as part of the subcontracted service. Therefore, all that is known is that it is guaranteed to have been instantiated when it might be needed, i.e., when the protocol flow has reached the subcontracted interaction pattern. As there is no mechanism yet to handle any delays that might occur if a subcontracted service cannot be instantiated immediately, probably the best default semantics for (sub)contracting is to instantiate sub-contracts at the same time the main contract is instantiated, and to start the main contract only after all sub-contracts have been established. For obvious reasons, this approach does not work for recursive service contracts!

Note that a sub-contract can never throw any contractual exceptions (see section 5.13.5), as protocols are obliged to handle such exceptions locally. It is allowed to throw a ProcessError or TimeoutError though, since these exceptions do not require a local handler. In this case, normal exception handling procedures apply.
Rationale

The decision to forbid a contract to throw contractual exceptions was made in order to prevent users from writing contracts that end by throwing around all kinds of un-handled exceptions. So if a task may throw an exception, the handling must be described by the contract, either in a formal handler or in an informal handler (see below). While an advantage for top-level contracts, this restriction might actually be too rigid if a contract is used as a subcontract. The problem is that a higher level contract is not aware of any exceptions thrown in the subcontract; when the latter ends, the higher contract can only assume that all requirements of the sub-contract were met. As a result, the burden that is placed upon a subcontract’s exception handling is severe: in effect, its handlers are still obliged to fulfill the original contract! The only other way out for a sub-contract is to give up shamefully by throwing a ProcessError.

For contract writers this means that a contract should always satisfy the same post-conditions, no matter if it managed to complete its regular processing or ended by executing an exception handler. Empty exception handlers are seldom adequate! If an exception handler ends with restart, retry, or resume, it does not actually end the contract, so in that case it has to guarantee the conditions that must hold at the point of continuation (see also section 5.13.5).

For contract users, it is advised to include relevant assertions immediately following a subcontracted interaction pattern. At least this helps to detect any problems at the earliest possible stage, and to prevent the parties from inadvertently executing subsequent tasks whose behaviour depends on the correct execution of the subcontract.

It is not unlikely that a future version of the LiNC languages has provisions to specify explicit pre- and postconditions on a contract as a whole, and that it allows contracts to throw exceptions. Of course it might still forbid such contracts to act as a top-level contract!

Some examples

The first example shows how a transport chain can be broken up into 3 separate sub-contracts: one to do the local transportation, one to transport the parcels by air, and one to do the local transportation at the remote end of the chain:

```plaintext
ordered sequential
  new transportLocal (provider, client, theParcels, theOrigin, theLocalSC);
  new transportAir (provider, client, theParcels, theLocalSC, theRemoteSC);
  new transportRemote (provider, client, theParcels, theRemoteSC, theDestination);
end sequential;
```

The provider and the client of all sub-contracts are the same, i.e., equal to the provider and the client of the current ‘master’ contract. The compiler verifies that the client and provider are legitimate actors for the subcontracted services. At each stage, a set of parcels is transported from a source to a destination location (service centers in most cases); this set is used as the service object of each of the sub-contracts, while the origin and destination at each stage are passed as service parameters. It would probably be wise to enclose the
sequential construct within a try construct (section 5.13.5) having a handler for possible ProcessErrors. Also, it is good practice to insert assertions (see section 5.13.10) before and after the sub-contract instantiations.

A second example shows the use of subcontracted interaction patterns inside a process block (see section 5.13.12). In this case the subcontracted services are delegated to a third party that is first set as the value of a service variable; the current provider is the client of both delegated services:

```
ordered sequential
  client : useSubcontractor (theSubcontractor);
  provider process
    ordered sequential
      new transportLocal (theSubcontractor, provider, theParcels);
      new transportRemote (theSubcontractor, provider, theParcels);
    end sequential;
  end process;
end sequential;
```

5.13.3 Selective interaction patterns

The selective interaction pattern specifies that exactly one of a set of interaction patterns can occur, but not which one. The selective interaction pattern is defined as follows:

```
selective_interaction_pattern ::= 'select' ( interaction_pattern ) + 'end' 'select' ';'
```

There is nothing at the protocol level to determine which of the nested interaction patterns will be executed; all that is specified is which nested patterns are allowed and which are not. The process logic of the contract parties is responsible for selecting the actual interaction pattern; at the protocol level the interaction thus seems nondeterministic. Any exceptions thrown from the nested patterns will be propagated to the context of the selective interaction pattern. So if a nested interaction pattern throws an exception, it is as if the selective interaction pattern itself had thrown the exception.

Rationale

The decision not to provide a means for controlling the choice and make the construct nondeterministic at the protocol level was taken after ample consideration. Whereas earlier publications (e.g. [ELL99]) referred to a special variable controlling the choice, the language at the time did not provide any way to actually specify or use the value of this control variable. Leaving it out therefore didn’t have any consequence for the contract designer and avoided the introduction of ‘magical’ elements into the language. It was also more in line with the grammatical view of protocols, which merely have to distinguish between valid and invalid interactions. Visible control variables do not contribute to this goal and, if usable at all, might tempt designers to putting too much control into a contract.
5.13. The Service Protocol

That having been said, we have indeed found examples of protocols where it would have been helpful to know which alternative was selected, and to use this knowledge to control another selection. Such a mechanism would have allowed for a significantly more compact protocol. Further research might reveal different mechanisms to accomplish the same goal in a way that is more in line with the current interpretation of protocols.

Some examples

This example is about a provider which will ship parcels on behalf of a client. The contract specifies that the provider has to provide two ways to ship parcels: either by surface (ship, train, etc.) or by air (plane):

```ml
select
  client: sendBySurface (clientParcel);
  client: sendByAir (clientParcel);
end select;
```

According to this protocol fragment, the client will issue exactly one command for each parcel. It is not known beforehand which of the two commands will be used to ship the parcel. A second example shows a protocol in which the two branches have different senders:

```ml
select
  client: sendBySurface (clientParcel);
  provider: sendByAirplane (clientParcel);
end select;
```

This protocol fragment specifies that either party could send a message. But it does not specify who is going to do it; this is left to the process logic of both contract parties. Obviously, the situation where both parties send a message is incorrect! In cases like these, the LINC implementation guarantees that only one alternative can be executed, namely the alternative whose task invocation message is received first (by the protocol verifier, that is). Of course, since network and other delays might be involved, there is always a chance that the second message is already on its way when the first message causes the protocol state to change. In this case the second message will result in a protocol violation. The implementation guarantees that the protocol verifier can never handle two messages at precisely the same moment, so there is always exactly one message that will be let through.

Ambiguity

The select construct is an example of a construct that may be ambiguous. In the following example, it will never be possible to determine which of the nested interaction patterns should be chosen in case a send () message is received:
select
  client: send (clientParcel);
end select;

One could argue that in this case it doesn't matter in this case which nested pattern is chosen. Of course it is only an extreme example, most cases are less obvious. Consider the following example:

select
  ordered sequential
    client: send (clientParcel);
    provider: confirmSentBySurface (clientParcel);
  end sequential;
  ordered sequential
    client: send (clientParcel);
    provider: confirmSentByAirplane (clientParcel);
  end sequential;
end select;

Again, there is no way to know which nested interaction pattern to select when the client issues a send () message; this would not be known until the provider replied with one of the confirmation messages. Though syntactically correct, this way to specify a protocol is considered bad style. Clearly, the intention of the protocol fragment should be expressed as follows:

ordered sequential
  client: send (clientParcel);
select
  provider: confirmSentBySurface (clientParcel);
  provider: confirmSentByAirplane (clientParcel);
end select;
end sequential;

Apart from being bad style, ambiguous protocols are to be avoided for another reason: in runtime, they will cause an AmbiguousProtocol exception to be thrown. A future version of the LinC compiler might detect ambiguous protocols during contract compilation.

5.13.4 Optional interaction patterns

The optional interaction pattern is used to specify that a certain interaction pattern may be executed once or skipped altogether. Its syntax is as follows:

optional_interaction_pattern ::= 'optional' interaction_pattern 'end' 'optional' ';'
A single interaction pattern may be specified inside the optional construct. There is nothing at the protocol level to determine if the nested interaction pattern will be executed. The process logic of the contract parties is responsible for this decision; at the protocol level the interaction thus seems nondeterministic. Any exceptions thrown from the nested pattern will be propagated to the context of the optional interaction pattern. So if the nested interaction pattern throws an exception, it is as if the optional interaction pattern itself had thrown the exception.

Rationale

The optional interaction pattern is not strictly necessary: any optional construct can be replaced by a selective interaction pattern containing two nested interaction patterns that are identical except for the first sub-clause of one of them: that sub-clause representing the optional part, of course. Using an optional construct avoids repetition and provides a much better expression of what is happening.

Some examples

The first example shows a truck requesting a platform to unload at. Because a platform might not be available right away, the provider may need to ask the client to wait until a platform is available. The wait request is embedded within an optional construct:

```plaintext
ordered sequential
  client: assignPlatform ();
  optional
    provider: waitAtParking ();
  end optional;
  provider: usePlatform (thePlatformNumber);
end sequential;
```

Alternatively, this example could have been formulated using a select construct by including two branches, the first one consisting of a `usePlatform()` message and the second one consisting of an ordered sequential containing both provider messages. The version using an optional construct is definitely easier to understand.

In the second example, the action following the optional construct is to be executed by a different party than the action inside the optional construct (this situation is somewhat similar to what we have seen in the select construct)\(^\text{12}\):

```plaintext
ordered sequential
  provider: payBill (theBill);
```

\(^\text{12}\)The example falls short of describing reality, since most non-virtual providers would not accept contracts describing their clients' payments as optional.
optional
  client: acceptPayment (theBill);
end optional;
provider: payBillNowOrElse (theBill);
end sequential;

Should the provider wait sending his last message until he thinks the client won’t send hers any more? Who knows? As a matter of fact, he can just try to execute his part. If the client happened to send her message after the provider decided to send his (but before he actually managed to do so), or if the provider happened to send his message just after the client decided to come into action, the protocol verifier will just throw a ProtocolViolation to the party that happened to be the last one that did something. The course of action is very much like a reminder to pay a bill: “If your payment has crossed this reminder, you may safely ignore this letter”.

Ambiguity

Just like the selective interaction pattern, the optional interaction pattern might be ambiguous. Ambiguity results if the construct following the optional interaction pattern matches the construct nested inside the optional interaction pattern:

ordered sequential
  optional
    client: acceptParcel (clientParcel);
  end optional;
  client: acceptParcel (clientParcel);
end sequential;

Admittedly the example looks a bit contrived; it would probably be the result of some editing mistake. Like before, the ambiguity might be ‘resolved’ at a later stage:

ordered sequential
  optional
    ordered sequential
      client: handleParcel (clientParcel);
    provider: rejectParcel (clientParcel);
  end sequential;
end optional;
  ordered sequential
    client: handleParcel (clientParcel);
  provider: acceptParcel (clientParcel);
end sequential;
end sequential;

The handleParcel () requests ask the provider to handle the given parcel. Now the provider may reject the parcel, but only the first time. If the client asks again, the provider
has to accept it. The provider may also choose to accept it in the first place. The example
can of course be reformulated to avoid the ambiguity.

5.13.5 Monitored interaction pattern

The monitored interaction pattern is used to handle exceptional cases that might originate
from the nested interaction pattern. It does so by specifying exception handlers that will
monitor the execution of any task invoked within the nested interaction pattern. As soon
as one of the monitored exceptions is thrown from a nested task, the nested interaction
pattern is abandoned and protocol flow continues with the handler for that exception. The
rules for a monitored interaction pattern are as follows:

\[
\begin{align*}
\text{monitored_interaction_pattern} &::= \text{`try'} \\
&\quad \text{interaction_pattern} \\
&\quad \text{exception_handlers} \\
&\quad \text{`end' `try'} ; \text{'} \\
\text{exception_handlers} &::= \text{exception_monitor} + \\
\text{exception_monitor} &::= \text{`when'} \text{exception_names} `; ' \text{exception_handler} \\
\text{exception_names} &::= \text{exception_name} ( `; ', \text{exception_name} ) * \\
\text{exception_name} &::= \text{`valid exception name'}
\end{align*}
\]

For each handled exception, an interaction pattern is included that will be executed in
order to recover from the exceptional situation. Any exceptions that might be thrown from
the interaction pattern but which are not handled, propagated to the context where they
can be treated as if they originated from the monitored interaction pattern itself. Also, if
an exception is thrown from the interaction pattern that is nested in one of the exception
handlers, it is treated as if it were thrown by the monitored interaction pattern itself.

Before we go in detail about the different exception kinds and the different ways to handle
exceptions, it should be stressed that the handling of exceptions in service contracts is of
extreme importance; even more so if a contract is intended to be used as a subcontract.
Since any contract may in fact be used that way, no contract is complete without the proper
exception handling!

Exception classes

The language distinguishes between three classes of exceptions:

- **Contractual exceptions** are exceptions that are defined and handled within the con-
  tract. They can be thrown from the process logic of any task that includes the
  exception in its throws clause (see section 5.12.1). The protocol is not allowed to end
  with an un-handled contractual exception, so any task that may throw one must be
  enclosed in a monitored interaction pattern that includes a handler for the exception.
Since they are explicitly defined in a task signature, we will often refer to contractual exceptions as named exceptions.

- **General exceptions** are exceptions that are defined by the LINC platform itself. Currently, there are two such exceptions: `ProcessError` and `TimeoutError`. They must not be declared in a task signature, even though they can be thrown from any task, either by the task's process logic or by the protocol verifier (in which case it is probably caused by a missing `validateReturn ()`, see section 5.12.1). General exceptions may or may not be handled by an exception handler: contrary to the case of contractual exceptions, the protocol is allowed to end by throwing one of these general exceptions.

- **Protocol exceptions** are also defined by the LINC platform itself. The exceptions of this class are limited to `ProtocolViolation`, `PreconditionViolation`, `PostconditionViolation`, and finally `AmbiguousProtocol`. They can only be thrown by the protocol verifier, and can never be handled in the protocol itself. Protocol exceptions are always thrown to the party that attempts to do something that violates the protocol; that party will have to correct the situation before the protocol can continue. Only the violating party will notice that something is wrong, the other parties are shielded from the mistake. It is an error to include a handler for one of these exceptions in a monitored interaction pattern.

Contractual exceptions will usually be handled by a **formal exception handler**, while general exceptions will usually be handled by an **informal exception handler** (see below).

### Rationale

The distinction between contractual exceptions and general exceptions reflects the different natures of the problems that might cause these exceptions to be thrown. Contractual exceptions correspond to situations that, though exceptional, may still be foreseen by the parties that agree on the contract. They form a list of things that might go wrong, while their handlers describe how parties agree to cope with the problems. In a way, the monitored interaction pattern functions as a kind of control structure here. General exceptions, on the other hand, represent truly exceptional cases: situations that are so remote from the normal flow of events that they cannot be listed; neither can their handlers. As we will see below, this also has an effect on the way these exceptions are treated by the exception handlers.

### Exception handler types

The language also makes a distinction between two different kinds of exception handler, i.e., **formal and informal**. Formal means that the exception handler is defined using regular protocol constructs which have to be followed after the exception has occurred. Informal means that the handling can not be defined within the contract, and that ad-hoc involvement of one or more actors is needed. An exception handler must either be specified as a
5.13. The Service Protocol

formal handler or as an informal handler; a combination of the two is not possible. The handlers differ in syntax, as shown by the following rules:

```plaintext
exception_handler ::= formal_exception_handler | informal_exception_handler
formal_exception_handler ::= interaction_pattern [ continuation_specification ‘;’ ]
informal_exception_handler ::= handler_invocation [ continuation_specification ‘;’ ]
handler_invocation ::= ‘to’ service_actor_list ‘;’ handler_invocation_message
service_actor_list ::= service_actor (’,’ service_actor )*
service_actor ::= ‘provider’ | ‘client’
handler_invocation_message ::= ‘valid task invocation message’
continuation_specification ::= ‘restart’ | ‘retry’ | ‘resume’ | ‘reject’
```

As shown by these rules, a formal handler consists of a regular interaction pattern, optionally followed by a continuation specification. This interaction pattern is not allowed to throw any contractual exceptions itself. An informal handler consists of a special kind of task invocation message that can also be followed by a continuation specification. The handler invocation message is special because it does not originate from a specific contract party, but from the contract logic itself; it is sent to all specified contract parties at once. Its arguments may be service objects (section 5.8), service parameters (section 5.9), and/or service variables (section 5.11). The handler task must be defined for all service actors of the contract; it is not allowed to throw any contractual exceptions, and may not have a precondition. A postcondition is allowed though, and may even be required to ensure that the handler leaves the protocol in a reasonable state\(^5\). The handler task terminates after all executing parties have completed their respective tasks.

**Rationale**

While there is a certain correspondence between contractual exceptions and formal handlers on the one hand, and between general exceptions and informal handlers on the other hand, this is not a requirement of the language. So a contractual exception might be handled by informal exception handler.

Both formal and informal exception handlers allow the contract to specify what should happen after the exception was dealt with. This is done by ending the exception handler with one of the following keywords:

- **restart** indicates that interaction must be rolled back to the very beginning of the contract,
- **retry** indicates that interaction must be rolled back to the beginning of the monitored interaction pattern,
- **resume** indicates that interaction must be rolled back to the beginning of the task invocation message that threw the exception,

---

\(^5\)This implies that an informal exception handler might still end by throwing a **ProcessError**.
• **reject** indicates that the entire contract should be aborted.

Absence of one of these keywords means that the protocol is resumed at the interaction pattern that follows the monitored interaction pattern. If the exception handler specifies that contract execution should be rolled back to an earlier state, the handler task should make sure that the internal state of each actor is also updated to reflect the roll-back.

_Rationale_

Continuation specifications extend the error handling capabilities of contracts considerably by allowing them to roll-back to a point where things were still alright. There is a certain risk involved, though, since errors have a tendency to show up long after the real harm was done. The motto should therefore be: “handle with care”.

_Some exception handling examples_

The first example presents the handling of a simple contractual exception. It is assumed that the provider’s `chargeParcels()` task might throw the exception `SetTooLarge`. This exception will be propagated by the sequential interaction pattern. But as the latter pattern is nested within a monitored interaction pattern, the exception can be caught by a formal handler:

```plaintext
try
  ordered sequential
    client : chargeParcels (theParcels);
    client : handleParcels (theParcels);
  end sequential;
when SetTooLarge :
  provider : failedHandlingParcels (theFailedParcels);
end try;
```

After the exception was thrown, the provider is obliged to send a `failedHandlingParcels()` message to the client, passing the set of parcels whose handling failed as an argument. Since there is no continuation specification, execution continues with the interaction pattern following the entire try construct. In the next example, a provider informs a truck driver to unload at a given platform; the contract is prepared to deal with drivers that manage to show up at the wrong platform:

```plaintext
try
  ordered sequential
    provider : usePlatform (thePlatformNumber);
    client : requestUnloading (theTruckNumber);
  end sequential;
when InvalidPlatform :
  retry;
end try;
```
5.13. The Service Protocol

When the driver requests unloading of the truck at the wrong platform, an InvalidPlatform is thrown at him. Since the formal handler consists of nothing but a retry, the entire sequence will have to be redone, starting with the provider telling the driver where to unload. No handler task is specified to inform the contract parties, since the named exception itself is already known to both parties and conveys enough information to enable them to roll back. A final example shows how an informal handler might be defined:

```
try
  ordered sequential
    provider : usePlatform (thePlatformNumber);
    client : requestUnloading (theTruckNumber);
  end sequential;
when SuspectedTruckLoad :
  to client. provider : clearDockingArea ();
  reject;
end try;
```

This example, which is a bit hypothetical, shows how a suspected truck load (e.g., some ticking, leaking, or otherwise suspected pallets) might be handled. As nobody can predict how such a truck load is to be handled, an informal handler is certainly appropriate. All that is known is that everybody who is not needed to handle the situation should leave the area, and that all parties who are involved should be notified of the situation. The contract will be rejected afterwards.

5.13.6 Timed interaction patterns

The timed interaction pattern is the only pattern that is directly concerned with the times at which interactions are attempted. Without this construct it would not be possible to include any restrictions to the (absolute or relative) times at which the interactions take place. The syntax of the timed construct somewhat resembles that of the monitored interaction pattern:

```
timed_interaction_pattern ::= 'timed' timing_specification
  interaction_pattern
  timing_exception_handlers
  'end' 'timed' ';'
timing_specification ::= '(' timing_constraints ')'
timing_constraints ::= timing_constraint ( ',' timing_constraint ) *
timing_constraint ::= absolute_constraint | relative_constraint
absolute_constraint ::= before_specification | after_specification | between_specification
between_specification ::= after_specification '!' before_specification
before_specification ::= 'before' time_expression
after_specification ::= 'after' time_expression
time_expression ::= service_parameter_reference | service_variable_reference | time_constant
```
time.constant ::= 'valid DateFormat constant'
relative.specification ::= duration.constraint | timeout.constraint
duration.constraint ::= 'duration' relative.constant time.unit.constant
timeout.constraint ::= 'timeout' relative.constant time.unit.constant
relative.constant ::= service.parameter.reference | service.variable.reference | positive.constant
positive.constant ::= 'positive integer or float constant'
time.unit.constant ::= 'valid time unit constant'
timing.exception.handlers ::= timing.exception_monitor +
timing.exception_monitor ::= 'when' timing.exception.names ':' exception.handler
timing.exception.names ::= timing.exception.name ( ',', timing.exception.name ) *
timing.exception.name ::= 'valid timing exception name'

Absolute timing constraints can be used to specify that the interactions must take place within some absolute time frame. The specified time values may be time constants or a reference to a service parameter (section 5.9), a service variable (section 5.11), or a component thereof. Time constants must be specified in correspondence to the default Java date and time format for the default locale (as defined by class DateFormat). In the case of a parameter, a variable, or a component, that parameter, variable, or component must be bound to a valid instance of class Calendar when the protocol flow reaches the timed interaction pattern. Furthermore, the fields of this instance must be set. Relative timing constraints are used to specify that the whole of the nested interaction must not exceed a specified duration, and/or that the delay between any two consecutive interactions must not exceed the given timeout value. Durations and timeouts are specified as a integer or float constants or references to service parameters (section 5.9), service variables (section 5.11), or components thereof. Service parameters, variables, or components must be bound to a non-zero integer or float value when the protocol flow reaches the timed interaction pattern. The constant value is followed by a time unit constant* defined in the public domain of the Logistic Contract Space (e.g., Second, Minute, Hour, Day, Week, Month, Year). Absolute and relative specifications can be combined in a single timed interaction pattern.

Rationale

Timing has not been part of the language until recently, apart from the timing constraints on multiple interaction patterns. It has long been recognized that the absence of timing constraints was a serious omission in some application areas. Instead of adding timing to each individual interaction pattern, it was considered best to design a single timing construct that could be combined freely with existing constructs. The timed interaction pattern described here was designed to be orthogonal with existing constructs.

Ultimately, all timing requirements apply to the actual task invocation messages occurring inside the timed interaction pattern, in particular to the beginning and ending times of those task invocation messages. The effects of violating a timing constraint depend on the

---

As parties should use the same clock to determine if their interactions are allowed, a common clock is made available via LincRemoteActor.
specific constraint that is being violated. For the absolute time frame constraint, both the starting time and the ending times of each nested task must be within the specified time frame. Ending time exceptions are not thrown until the task ends; running tasks will not be interrupted when a time constraint is about to be violated. The relative duration constraint will cause a DurationError exception if the specified duration has been exceeded when a nested task invocation ends; the relative timeout constraint will cause a TimeoutError when the delay between the end of a task and the start of the next one exceeds the specified timeout value.

Any timed interaction pattern containing a before and/or after constraint must specify a handler for the TimeFrameError exception. Likewise, the presence of a duration constraint and/or a timeout constraint dictates the presence of a handler for a DurationError and a TimeoutError, respectively. Although the latter exception is in fact the same exception as the TimeoutError we saw in section 5.13.5, the exception as described here cannot propagate out of the timed interaction pattern and will be treated as if it were a different exception.

**Rationale**

Timing errors are really a subclass of protocol errors, and it would not be difficult to treat starting time violations as such. However, it would be quite impractical to deny a contract party the right to complete a task because the specified ending time limit has passed: even though it is clearly a violation of the protocol, throwing a ProtocolViolation to the offender would only make matters worse. The only reasonable way out is to let the task end and throw an exception as well. The exception must be handled though, and not be allowed to propagate beyond the timed interaction pattern, hence the obligation to include the appropriate handler(s); if it were not handled, the protocol violation might go unnoticed. It is for reasons of consistency that starting time violations are treated the same way.

**An example**

The following example specifies that a house is to be painted before a given date, and be completed within one week. In case of a violation of one of these constraints the provider has to offer a compensation to the client:

```plaintext
timed (before May 1 2003, duration 1 Week)
    client : paintHouse (myHouse);
when TimeFrameError :
    // Job not accomplished within specified time frame
    provider : acceptCompensation ();
when DurationError :
    // Job exceeded specified period
    provider : acceptCompensation ();
end timed;
```
5.13.7 Sequential interaction patterns

The *sequential interaction pattern* is used when a number of tasks must be executed in a non-overlapping way. There are two variants of the construct: *ordered* and *unordered*. The ordered sequential construct is more common; it specifies that the nested constructs should be executed in the specified order, without any overlap between a given task and its successor. A *ProtocolViolation* exception will be thrown if these conditions are not met. The unordered variant merely specifies that the nested constructs should be executed in a non-overlapping way; nothing is said about the order in which they are executed. The syntax of a sequential interaction pattern is given by the following rules:

```plaintext
sequential_interaction_pattern ::= sequential_modifier 'sequential'
                           ( interaction_pattern | sequential_action ) +
                           'end' 'sequential' ';'
sequential_modifier ::= 'ordered' | 'unordered'
sequential_action ::= update_construct | assertion_construct | process_block
```

The sequential interaction pattern demands that each nested interaction pattern has completed before another interaction pattern in the sequence can start. This means that the sending of a task invocation message that belongs to a certain nested interaction pattern must be delayed until all tasks being executed under the preceding nested interaction pattern have terminated\(^{15}\). Execution of the sequential interaction pattern terminates when the last of its nested interaction patterns terminates. If a task executed from within a nested interaction pattern throws an exception which is not handled locally, it is as if the exception were thrown by the sequential interaction pattern itself.

**Rationale**

The two variants were designed this way in order to make the ordering aspect orthogonal to the sequential, non-overlapping aspect. Note that a variant could be envisaged that merges some of the aspects of a sequential interaction pattern with that of concurrent execution: such a construct would allow a task invocation message to be executed under a new interaction pattern to be sent before the previous nested interaction pattern has ended. This variant could be implemented in a way that automatically serializes the messages so that they do not overlap. The sender would not have to care anymore if it is safe to send a message; it would just appear to him as if it took somewhat longer to execute the task. A variant like this was rejected because it would be a departure from the semantics of all other protocol constructs, which only check conditions and do not interfere with the parties' process logic.

The sequential interaction pattern is the only protocol construct that can contain nested *sequential actions* as well as nested interaction patterns. Sequential actions are constructs

\(^{15}\)It is up to the sender to determine if he can safely send a new task invocation message!
that check or manipulate service objects (see section 5.13.10 and section 5.13.11), or that describe some aspects of the internal process taking place in one of the contract parties (see section 5.13.12).

**Rationale**

The reason to restrict assertions and updates to sequential interaction patterns is that such constructs only make sense at specific points of execution of the contract; they should reflect or modify the state of the contract at that point in time. Sequential constructs are the only constructs that have such a notion of time; even though this notion is relative to the beginning and/or ending of tasks. (Concurrent interaction patterns may contain a concurrent action, which does not allow for assertions or updates).

It might be the case that all tasks are to be executed by the provider, at times that are determined by the client. The client then signals the start of each new task to be executed by sending a task invocation message. It might also be the case that both the provider and the client have to take turns in executing tasks. This arrangement can be used to implement a simple ‘handshaking’ protocol.

**Some examples**

In the first example, the client sends all the messages to the provider, which then has to execute a task in response to each message:

```plaintext
ordered sequential
  client: announce (clientParcels);
  client: enter (clientParcels);
  client: clear (clientParcels);
end sequential;
```

If the execution of announce () would throw an exception, enter () and clear () would never be executed. However, if enter () would throw an exception, only clear () would never be executed. The next example shows a simple handshaking protocol, in which the sending of each new message has to be explicitly granted by the provider. It does so by sending a message to the client each time it has completed a task (i.e., readyForNewCommand ()), to let the client know it is ready to receive a new task invocation message. The tasks that are implemented by the client do not have to accomplish anything; they are just used as signals then provide information about the provider’s internal state.

```plaintext
ordered sequential
  client: announce (clientParcels);
  provider: readyForNewCommand ();
  client: enter (clientParcels);
  provider: readyForNewCommand ();
```
client: clear (clientParcels);
provider: readyForNewCommand ();
end sequential;

There might be circumstances where this is useful, e.g., where the client produces task invocations from within different threads. In such a case the client may not know if it is safe to send a new task invocation message. The completion messages from the provider might come in handy under such circumstances.

Ambiguity

Ambiguity might be an issue with sequential interaction patterns, depending on the ordering constraint. Obviously, tasks should be executed one by one in an ordered construct; there is never any doubt which task should be started when a task invocation message is received. Matters may be different with the unordered sequential construct though. Consider the following example:

unordered sequential
   client: announce (clientParcels);
   client: enter (clientParcels);
   client: enter (clientParcels);
end sequential;

As any task invocation message might be the first one to be executed, sending an enter () message is ambiguous here. Although this is something to be aware of, real protocols are not likely to contain constructs like this one.

5.13.8 Concurrent interaction patterns

The concurrent interaction pattern is used when a number of tasks must be executed that can be run independently from each other. Just as with the sequential interaction pattern, all nested interaction patterns are to be executed. The difference is that the nested interaction patterns can be executed concurrently; the contract parties don’t have to wait until a certain interaction pattern has completed before they can start executing tasks within other nested interaction patterns. There are three variants of the construct: ordered, nested, and unordered. The ordered concurrent interaction pattern specifies that a number of nested interaction patterns must be started one after another in the order in which they are specified; a ProtocolViolation exception will be thrown if this constraint is violated. A nested concurrent interaction pattern specifies that the nested interaction patterns must be started one after another in the order in which they are specified, and furthermore, that they must end in exactly the opposite order. The unordered concurrent interaction pattern specifies that a number of nested interaction patterns can start and terminate in any order. In all
variants, execution of the nested interaction patterns may overlap\(^\text{16}\). The syntax of the concurrent interaction pattern is specified as follows:

\[
\text{concurrent}\_\text{interaction}\_\text{pattern} \\
::= \text{concurrent}\_\text{modifier} \text{`concurrent'} \\
\text{( interaction}\_\text{pattern} | \text{concurrent}\_\text{action} ) + \\
\text{`end'} \text{`concurrent'} ';'
\]

\text{concurrent}\_\text{modifier} ::= \text{`ordered'} | \text{`unordered'} | \text{`nested'}

\text{concurrent}\_\text{action} ::= \text{process}\_\text{block}

The concurrent interaction pattern terminates when the last of its nested interaction patterns terminates. If one of the nested interaction patterns throws an exception, the remaining nested interaction patterns are allowed to complete. Still, if the exception is not handled locally, it will be propagated, and it is as if the exception were thrown by the concurrent interaction pattern itself.

\textit{Rationale}

While the ordered sequential interaction pattern is the most natural of the various sequential constructs, the most natural of the concurrent constructs is the unordered variant. The ordered variant is supplied for completeness' sake, and the nested variant is a special case that is especially useful to describe delegation of tasks to third parties. Delegation is really an aspect of the implementation of a task; sometimes it is useful to define and/or expose part of an implementation in the form of a process block (see Section 5.13.12). An important part of delegation is of course that a party that delegates some parts of its task to third parties cannot tell its own client that the task was completed before its sub-suppliers have completed their respective jobs. Therefore, delegation can be described by a nested concurrent construct consisting of two parts: the first part is the task invocation message requesting the provider to do something; the second part is a provider process block containing one or more subcontracted services. See also the examples below.

The concurrent interaction pattern is the only protocol construct that can contain nested \textit{concurrent actions} as well as nested interaction patterns. Concurrent actions are constructs that describe some aspects of the process taking place in one of the contract parties (see Section 5.13.12).

\textit{Some examples}

The following example shows how a concurrent pattern might be used within a sequential pattern. It describes the assembling of a car from components that have been manufactured in a number of independent subprocesses.

\(^{16}\text{An additional variation may be envisaged that has a parameter to limit the number of tasks that may be executed in parallel, e.g., unordered concurrent (3). For now, the need of this feature has not been determined.}\)
ordered sequential
  unordered concurrent
    client: constructChassis();
    client: constructEngine();
    client: constructInterior();
  end concurrent;
    client: assembleCar();
  end sequential;

Note that the tasks of manufacturing the parts may be started in any order. Because
the concurrent construct won’t end before all the nested patterns have been completed, the
assembly stage can safely follow the concurrent construct in an enclosing ordered sequential
interaction pattern. The next example presents a typical example of a nested concurrent
construct:

  nested concurrent
    client: unloadTruck();
    provider process
      new truckUnloading (any, provider);
    end process;
  end concurrent;

In this case it is not the provider who unloads the truck but some unidentified third party (as
specified by any). The current provider is the client of the delegated service. This fragment
describes one thing we know beforehand about the implementation of the unloadTruck ()
task: that any subtask started on behalf of the task will only start after the calling task
was started, and that the subtask will end before the calling task ends.

Ambiguity

Ambiguity might be an issue with concurrent interaction patterns like it was with the se-
quential interaction pattern, but the risk of running into an ambiguous situation is definitely
larger. It is easy to see that this is the case as now there may be several branches that are
active at the same time, not just one. A new task invocation message could in principle be
executed in each of the active branches:

  ordered concurrent
    ordered sequential
      client : prepareForSomething ();
      client : doSomething ();
    end sequential;
    ordered sequential
      client : prepareForSomethingElse ();
      client : doSomething ();
    end sequential;
  end concurrent;
Ambiguity would occur if the task `doSomething()` were invoked after both `prepareForSomething()` and `prepareForSomethingElse()` had finished. If only one of these tasks had finished, there would be only valid way to execute `doSomething()`, so the ambiguity would be avoided.

### 5.13.9 Multiple interaction patterns

The *multiple interaction pattern* specifies that a single nested interaction pattern should be executed a certain number of times. All these instances of the nested interaction pattern can be run either sequentially or concurrently, depending on whether the keyword *sequential* or the keyword *concurrent* is added as a modifier to the multiple interaction pattern. Execution of a sequential multiple follows the same rules as execution of an ordered sequential construct containing several identical interaction patterns; execution of a concurrent multiple mirrors execution of an ordered concurrent construct containing several identical interaction patterns. The multiple interaction pattern is defined as follows:

```
multiple_interaction_pattern
  ::= [ multiple_index ]
      multiple_modifier 'multiple' multiple_constraint
      interaction_pattern
      'end' 'multiple' ';'

multiple_modifier ::= 'sequential' | 'concurrent'
multiple_constraint ::= definite_constraint | indefinite_constraints
definite_constraint ::= (' number_constraint ')number_constraint
  ::= service.parameter.reference | service.variable.reference | positive.constant
positive.constant ::= 'positive integer constant'
indefinite_constraints
  ::= [ limit.constraint ]
      [ duration.constraint ]
      [ exit.constraint ]
```

Since multiple interaction patterns are usually ambiguous, it is often required to add a *multiple_index* to the multiple. This index appoints a common task argument as an index that groups task invocation messages together (see below). There are two forms of the multiple interaction pattern: *definite* and *indefinite*. In the definite form, a *number constraint* is specified that determines the exact number of instances of the nested interaction pattern that should occur. This number must be an integer constant or a reference to an integer valued service parameter (section 5.9), service variable (section 5.11), or component thereof. The service parameter, variable, or component must be bound to a positive (non-zero) value when the protocol flow reaches the multiple interaction pattern. In the indefinite form, the number of nested interactions is determined by a set of limit, duration, and/or exit constraints. It is also possible to omit all constraints, in which case the multiple interaction pattern will continue to create instances of the nested interaction pattern. Note that the multiple constraints do not affect termination of the multiple interaction pattern. Instead,
they mark the construct as closed, thereby prohibiting the creation of new instances of the nested interaction pattern.

Since several constraints can be specified for the multiple interaction pattern, where each constraint can have a minimum and/or a maximum value, it is necessary to have a precise definition of their interaction. First we need to distinguish between implicit closing and explicit closing of the construct: implicit closing is the closing enforced by one of the constraints; explicit closing is closing by means of a quit () message. The following rules determine how the various constraints interact:

- If one or more of the minima are not yet satisfied, the multiple cannot be closed explicitly. If one of the parties issues a quit () before all minima are satisfied\textsuperscript{17}, a ProtocolViolation will be thrown to the offending party. If all minima are satisfied, the multiple can be closed explicitly by means of a quit () message.

- If one or more of the maxima are reached, the multiple will be implicitly closed, even if some minimum constraint has not yet been satisfied. Any task that will be invoked after a maximum was reached will be interpreted in the protocol construct that follows the multiple interaction pattern.

Minimum constraints are therefore relevant for explicit closing, while maximum constraints determine implicit closing if the multiple has not been explicitly closed before. In all cases, the effect of closing a multiple construct is that no new instances of the nested interaction pattern will be created. The existing instances are allowed to terminate, even if this would mean that new tasks will be started after the multiple interaction was closed.

**Rationale**

As will be shown below, all indefinite variants of the multiple interaction pattern allow one to specify both a minimum and a maximum constraint on the creation of new instances of the nested interaction pattern. Interpretation of the maximum constraints is straightforward: no new nested branches will be created after a certain condition has become true; the next interaction that would imply the starting of a new branch will be interpreted as belonging to the construct that follows the multiple interaction pattern. The minimum constraint can be interpreted in more than one way though. One possible interpretation would be that parties can escape from the multiple at any time after the minimum was reached, just be sending a message that cannot be executed in the context if the multiple (but that can be executed in the context of the interaction pattern following the multiple). Another interpretation is possible if a multiple can only be ended by reaching a limit or by means of one of the parties invoking quit (). In that case, a minimum constraint could be used to deny a party the right to end the multiple before all minimum constraints are met\textsuperscript{18}. It is this second interpretation that we have chosen.

---

\textsuperscript{17}Or if one of the parties invokes a task that can only be executed in the protocol construct that follows the multiple, if this extension might ever be implemented.

\textsuperscript{18}It is a topic for debate if the minimum constraints should also interact with the maximum constraints, or if they should only deny explicit closing by means of quit ().
The current implementation of the LINC compiler requires one of the contract parties to send a \texttt{quit (}) message if the multiple is to be ended before one of the maximum constraints is violated. This requirement might well be lifted someday, allowing for termination of a multiple by sending a message that can only belong to the construct that follows the multiple; as long as \texttt{quit (}} remains supported, this does not pose a problem for the process logic of existing parties, nor for existing contracts that are supported by them.

The \textit{limit constraint}

The \textit{limit constraint} is a constraint on the number of branches allowed in the multiple interaction pattern. New branches will only be created if the constraint is satisfied; existing branches are allowed to complete even after the specified number of branches has been reached, even if this would mean that new tasks will be executed in those branches. The limit constraint can specify a minimum value, a maximum value, or both. If both values are specified, the minimum value must not exceed the maximum value. The syntax of the limit constraint is as follows:

\[
\begin{align*}
\text{limit.constraint} & ::= (' \text{limit'} : \text{limit.specification} ') \\
\text{limit.specification} & ::= \text{minimum.limit} | \text{maximum.limit} | \text{minimax.limit} \\
\text{minimum.limit} & ::= '\text{minimum'} \text{limit.value} \\
\text{maximum.limit} & ::= '\text{maximum'} \text{limit.value} \\
\text{minimax.limit} & ::= \text{minimum.limit} ',' \text{maximum.limit} \\
\text{limit.value} & ::= \text{service.parameter.reference} | \text{service.variable.reference} | \text{limit.constant} \\
\text{limit.constant} & ::= '\text{positive integer constant}'
\end{align*}
\]

The minimum value specifies that the multiple cannot be explicitly closed (i.e., by a \texttt{quit (}) if the number of instances is less than the given lower bound; it can be closed by one of the maximum constraints though. The maximum value specifies that the number of instances cannot exceed the upper bound: the multiple will be closed as soon as the maximum is reached. The values used as a minimum or maximum limit must be integer constants or references to integer valued service parameters (see section 5.9), service variables (see section 5.11), or components thereof. Service parameters, values, or components must be bound to (non-zero) values when protocol flow reaches the multiple interaction pattern.

\textit{Rationale}

Even though the effect of the definite variant could be obtained by means of the indefinite variant with the appropriate limit constraint, the definite variant is included in the language since it more clearly expresses the intent of this simple variant.
The duration constraint

The duration constraint specifies a minimum and/or maximum duration for the multiple interaction pattern. The duration is counted from the start of the first task invocation message in the multiple interaction pattern. New branches will only be created if the constraint is satisfied; existing branches are allowed to complete even after the specified duration has passed, even if this would mean that new tasks will be executed in those branches. The duration constraint can specify a minimum duration, a maximum duration, or both. If both values are specified, the minimum duration must not exceed the maximum duration. The syntax of the duration constraint is as follows:

\[
\begin{align*}
\text{duration.constraint} & ::= (\text{'duration': duration.specification} ) \\
\text{duration.specification} & ::= \text{minimum.duration | maximum.duration | minimax.duration} \\
\text{minimum.duration} & ::= \text{'minimum'} \text{ duration.value time.unit.constant} \\
\text{maximum.duration} & ::= \text{'maximum'} \text{ duration.value time.unit.constant} \\
\text{minimax.duration} & ::= \text{minimum.duration}, \text{ maximum.duration} \\
\text{duration.value} & ::= \text{service.parameter.name | service.variable.name | duration.constant} \\
\text{duration.constant} & ::= \text{'positive integer or float constant'}
\end{align*}
\]

The minimum duration specifies that the multiple cannot be explicitly closed (i.e., by a quit \( () \)) if the elapsed time is less than the given lower bound; it can be closed by one of the maximum constraints though. The maximum duration specifies that the duration of the multiple cannot exceed the given period; the multiple will be closed as soon as the maximum duration is reached. The values used as a minimum or maximum duration must be integer or float constants or references to integer or float valued service parameters (see 5.9), service variables (see 5.11), or components thereof. Service parameters, values, or components must be bound to (non-zero) values when protocol flow reaches the multiple interaction pattern. Duration values are followed by a time unit constants defined in the public domain of the Logistic Contract Space (e.g., Second, Minute, Hour, Day, Week, Month, Year).

Rationale

One might wonder why there is something like a duration constraint if one could use a multiple interaction pattern inside a timed interaction pattern. The main reason is that the latter combination would force the user to end the multiple by means of a quit \( () \) before the maximum duration of the multiple was reached. Without this quit \( () \), the timed interaction pattern would throw a DurationError when the period was over, a situation to be avoided since exceptions are not intended to be used as a control flow mechanism.

The exit constraint

The exit constraint declares an exit-label that contract parties can use as an argument to a quit \( () \) task invocation message (see below). The syntax of the exit constraint is as
5.13. The Service Protocol

follows:

\[
\begin{align*}
\text{exit.constraint} & ::= \text{`(exit)`} \text{`;`} \text{simple.exit.label.name} \\
\text{simple.exit.label.name} & ::= \text{`unique exit label`} \\
\end{align*}
\]

The name to be used as an exit label must be new and unique within the scope of the contract. Obviously, the task signatures of the receiving party must include a `quit()` task signature (see section 5.12.1). New branches can only be created if the constraint is satisfied, i.e., until one of the parties has called `quit()`; existing branches are allowed to complete even after the `quit()`, even if this would mean that new tasks will be executed in those branches. The syntax of a `quit()` task invocation message is identical to that of a regular task invocation message; the exit label name of the multiple to be ended must be passed as a parameter to `quit()`. As ending a multiple does not force its branches to an end, quitting an outer multiple does not imply the ending of possible inner multiples! As a result, each individual nested multiple must be explicitly or implicitly ended.

**Multiple indices**

As shown by the above rules, the multiple interaction pattern may also contain a *multiple index*. Its syntax is as follows:

\[
\begin{align*}
\text{multiple.index} & ::= \text{`indexed`} \text{`(index.name)`} \text{`:`} \text{index.type`} \\
\text{index.name} & ::= \text{simple.index.name} \\
\text{simple.index.name} & ::= \text{`unique name`} \\
\text{index.type} & ::= \text{name} \\
\end{align*}
\]

As one can see from these rules, the multiple index is actually a kind of declaration. Index names must be unique within the scope of the current contract. Index types must be defined as types in the Logistic Contract Space (section 5.4); the corresponding classes must implement the Serializable interface. Multiple indices may be used as arguments in task invocation messages (see section 5.13.1), as service parameters or service indexes to subcontracted services (section 5.13.2), or as arguments in assertions (section 5.13.10) or updates (section 5.13.11). Service indices can also be used in the pre- and postconditions of task signatures (section 5.12.1).

In the absence of an index, the multiple invocation pattern merely specifies that a certain set of patterns is to be executed sequentially or concurrently. If the index is present, any task invocation message that is physically nested within the multiple is required to include an argument with the name and type specified by the index; this argument will be referred to as the index argument. The absence of an index argument is marked as an error:**

\[\text{\textsuperscript{19}Note that this might require an argument to be added to the task signatures. Quite often though, the task invocation messages will already contain an argument that can be used as an index argument quite naturally! In such a case it still might be necessary to specify a name for the argument in the task signature (in case a name is not already present).}\]
Each individual task invocation message can now be attributed to a specific instance of the multiple. Note that a task invocation message that is nested in more than one multiple interaction pattern must have an index argument for each of them. It must also have index arguments for any service indexes passed from the context of the contract (see section 5.10).

**Rationale**

It should be easy to see that concurrent multiple interaction patterns are generally ambiguous *by definition*: only the very simple case of a nested construct that consists of a single task invocation message is not. The reason is of course that if the nested construct consists of a single task invocation message, then each new task invocation message can only be interpreted as a new instance of the nested interaction pattern. If the nested construct consists of anything else than a single task invocation message, any particular task invocation message could in principle belong to any existing branch of the multiple or even to a new one. Index arguments solve this problem, since they restrict execution of a task to the branch that corresponds to the actual index value.

In the simple case of a single task invocation message nested within the multiple (no matter if it is sequential or concurrent), the absence of an index argument would not pose a major problem for the client and the provider: they just have to generate and handle a set of task invocation messages in the order in which they occur. Even if there would be more than one message with the same argument value(s), only one interpretation would be possible: every message gets its own branch. But if the multiple should contain, e.g., a sequential interaction pattern, things would start to be different for a sequential multiple vs. a concurrent one: a sequential multiple still demands that the nested sequentials are executed one by one: there is no ambiguity. The concurrent multiple, in contrast, allows for overlap of the nested sequentials. In this case it would be impossible for the message receiver to know to which particular nested sequential the message belongs, unless it can use the value of an index argument to 'assign' the message to a specific branch of the multiple. Without this multiple index, the multiple would be very ambiguous indeed. Of course this requires that the values supplied for the index argument by the sender are appropriate: it must be possible to divide the overall set of task invocation messages into a number of subsets, where each subset corresponds to a specific index value.

When an indexed multiple contains a subcontracted interaction pattern, the semantics of both constructs require the multiple index value to be passed as a service index to the subcontracted service (see also section 5.10).

**Rationale**

The protocol of the subcontracted service is in fact substituted for the subcontracted interaction pattern, so any task invocation message sent in the nested protocol must still be verified in the context of the 'master' protocol. So if the master protocol contains a multiple construct, the message must be assigned a branch of the multiple, even though it will be passed on to the sub-protocol for further verification. Since the compiler needs to generate errors in case some task invocations messages do not contain the necessary
index arguments, contracts that must be usable as a sub-contract must have an indexes section to pass possible multiple indexes from the context.

The need to pass indexes from one contract to another is quite undesirable; it would be better if contracts could be decoupled from each other, and if task invocation could be interpreted in just the context of the contract in which they are ultimately executed. Further research is needed to see if it is possible to eliminate the indexes section from service contracts.

Using variables within multiples

A multiple interaction pattern specifies a single nested interaction pattern that acts as a template to create instances of that interaction pattern. Index arguments are used to assign task invocation messages to instances of the nested pattern. The possible values of these index arguments are determined by the type that was specified for the multiple index.

Tasks that are invoked in a multiple can have more arguments than just index arguments, of course. The protocol uses regular service objects, service parameters, or service variables for those non-index arguments. If such a service object, parameter, or variable would be of a simple type, the actual value passed for a non-index argument would have to be the same in each branch of the multiple! This is due to the binding mechanics of objects, parameters, and variables: when unbound, they are set by the first message that refers to them; when bound, the new value must be equal the existing value (see section 5.11). If the values have to be different in each branch, the variables corresponding to non-index arguments can be defined as maps, indexed by the multiple index. The net effect of this is a fresh variable (parameter, object) for each branch of the multiple, which is exactly what is intended in most cases.

Rationale

As argued in section 5.4 there are actually 2 solutions to this problem. Apart from map variables, one could also imagine a way to declare local variables that are instantiated once for each instance of the multiple. The main advantage of local variables would be that the syntax to refer to them would be simpler; the disadvantage is of course that a way to declare local scopes would have to be introduced to the language.

Ambiguity

As we have seen above, concurrent multiple interaction patterns would be hopelessly ambiguous without the help of indexes. E.g., the following example is always ambiguous, even though the sequential construct is ordered:
concurrent multiple (50)
  ordered sequential
    client: announce (clientParcel);
    client: enter (clientParcel);
    client: clear (clientParcel);
  end sequential;
end multiple;

Not only would the construct be ambiguous, there would be an additional problem: all task invocation messages under this multiple would have to pass the same argument value! This is clearly not the intention of the contract. Both problems are fixed by means of a multiple index:

indexed (clientParcel : Parcel)
concurrent multiple (50)
  ordered sequential
    client: announce (@clientParcel);
    client: enter (@clientParcel);
    client: clear (@clientParcel);
  end sequential;
end multiple;

By adding an index to the multiple, each task invocation message is required to have an argument that can be used by the provider to distinguish between different instances of the nested interaction pattern. This index is used for protocol verification only; it is still the responsibility of the receiving party’s business logic to regroup the messages belonging to each instance.

Some examples

The first example shows the same sequential group of task invocation messages we have seen before. It is enclosed within a multiple interaction pattern which specifies that exactly 50 of those sequences will occur, and that all of these sequences may be executed concurrently:

indexed (theParcel : FreightUnitTransport.Parcel)
concurrent multiple (50)
  ordered sequential
    client: announce (@clientParcel);
    client: enter (@clientParcel);
    client: clear (@clientParcel);
  end sequential;
end multiple;

The following example shows how the number of instances can be limited to at most 50:
5.13. The Service Protocol

indexed (theParcel : FreightUnitTransport.Parcel)
concurrent multiple (limit : maximum 50)
  ordered sequential
    client: announce (@clientParcel);
    client: enter (@clientParcel);
    client: clear (@clientParcel);
  end sequential;
end multiple;

If used like this, the multiple interaction pattern would never terminate if the client should send less than 50 announce () / enter () / clear () message sequences! So obviously it is more useful to include a duration or timeout value as well:

indexed (theParcel : FreightUnitTransport.Parcel)
concurrent multiple (limit : maximum 50) (duration : maximum 12 Days)
  ordered sequential
    client: announce (@clientParcel);
    client: enter (@clientParcel);
    client: clear (@clientParcel);
  end sequential;
end multiple;

The next example shows how a quit () message can be used to escape from the multiple. The multiple construct itself introduces a label to be referred to in the quit () message. In this case, it is the client which is allowed to close the multiple construct. Of course, the provider could also be allowed to close the multiple: it is allowed to include more than one quit () message\textsuperscript{20}.

indexed (theParcel : FreightUnitTransport.Parcel)
concurrent multiple (exit : noMoreParcels)
  select
    ordered sequential
      client: announce (@clientParcel);
      client: enter (@clientParcel);
      client: clear (@clientParcel);
    end sequential;
    client: quit (noMoreParcels);
  end select;
end multiple;

It is possible to exit an outer multiple, i.e., a multiple that is not the innermost enclosing multiple of the quit () message. When an outer multiple is closed, that does not mean that multiples inside that multiple will also be closed: they will accept new interaction patterns until they are closed themselves; it is only the specified multiple that will not create any new instances of its nested interaction pattern.

\textsuperscript{20} Just like there may be more than one party that can send a quit () message, there might be more than one party that receives it. This would be the case in multi-party contracts (see section 5.16).
indexed (theParcel : FreightUnitTransport.Parcel)
concurrent multiple (exit : thatsIt)
    concurrent multiple (exit : noMoreParcels)
    select
        ordered sequential
            client: announce (@clientParcel);
            client: enter (@clientParcel);
            client: clear (@clientParcel);
        end sequential;
        client: quit (thatIts);
    end select;
end multiple;
end multiple;

5.13.10 Assert constructs

The assert construct is different from previous protocol constructs in that it is not an interaction pattern. Instead, it is a protocol level construct used to state some conditions that must hold at a certain point of the protocol. There are two forms of the assert construct:

assert.construct ::= '{' 'assert' boolean.expression ';' '}'
assert.construct ::= 'assert' assertions 'end' 'assert' ';
assertions ::= ( boolean.expression ';' ) +

The latter form is more readable if the number of assertions is large, while the former is better if only a single condition must be checked. The boolean expression(s) specified in the assert construct must be expressions operating on a service object, i.e., whose target is the service object, or a component thereof. The service object (or component) must be bound at the time the assertion is evaluated. The same is true for any arguments used in the expression. The syntax of the boolean expressions is identical to the syntax of Java expressions; the Java operators &&, ||, and ! may also be written as and, or, and not, respectively. An expression may refer to service objects (section 5.8), service parameters (section 5.9), service indexes (section 5.10), temporary variables (section 5.11), and multiple indexes (section 5.13.9). If all expressions evaluate to the value true, execution will continue normally. If an expression evaluates to the value false however, the AssertionError exception will be thrown by the first task invocation message that is executed after the assertion check took place. This will also happen when the service object is not bound or if any service objects, service parameters, service indexes, or temporary variables used as arguments in the expression are not bound.

Rationale

The reason that the AssertionError exception is not thrown by the failing assertion itself is that the mechanics of exceptions require them to be thrown from a certain
task to the invoker of that task. But the assertion is unrelated to the tasks defined by the contract, and the invoker of the assertion test is the protocol manager (which is hidden from the parties executing the contract). Although it would be possible to have the protocol verifier send a notification message of the failing assertion to all contract parties, a real exception was preferred, and it was decided to treat the assertion check as a special kind of precondition on all subsequent task invocations. The only alternative, i.e., to treat it as a postcondition on the preceding task invocation, cannot be realized because in some protocol constructs one cannot know which task invocation will be the last.

Since assertions must be tied to a specific point in time at which they should hold, they can only be specified in ordered sequential constructs, where they can precede and/or follow regular interaction patterns. The assertions are checked after the preceding task invocation message has completed, but before the following task invocation message will start: the exact time at which they are checked is left unspecified.

Rationale

Assertions are conditions that are assumed to hold at a certain recognizable state of the protocol. With ‘recognizable state’ we mean a moment in time that follows the execution of some sub-protocol and precedes execution of a subsequent sub-protocol. Such a moment can only exist between 2 task invocations in a sequential construct. Note that this does not necessarily mean that no tasks are active at the time of the check, as the sequential construct might itself be embedded in some concurrent construct.

Although assertions are currently restricted to ordered sequential interaction patterns, some future version of the language might allow them to be included in sequential multiple interaction patterns as well.

Note that the assertions (just like task pre- and postconditions, for that matter) may refer to methods that are defined for objects that are referred to by the contract but not defined as part of that contract. This means that the validity of those methods can not be established (and hence the validity of the assertion, pre- or postconditions), and that the contract parties have to accept them for what they are worth. It is therefore very important that the classes that are available from the Logistic Contract Space are open-source and read-only, so that every party can verify them before engaging in the execution of the contract.

Some assertion examples

The following assertion shows how one could assert, in a contract concerned with unloading of a truck, that all pallets that were originally in the truck will be in the platform stack afterwards:

```java
{ assert thePlatform.getPlatformStack().getPallets() == theDropList.getPallets(); }
```
Both thePlatform and theDropList refer to service objects of the contract. Service variables and parameters can be used as arguments to methods that are part of the assertion, but not as the object the assertion is operating upon. A second example shows how one could use assertions to specify the conditions that should hold before and after the instantiation of subcontracted interaction patterns:\footnote{21If the language were extended to allow pre- and postconditions on contracts, there would be no need for such assertions!}

```plaintext
ordered sequential
  { assert theParcels.getLocation () == theOrigin; }
new transportLocal (provider, client, theParcels, theOrigin, theLocalSC);
  { assert theParcels.getLocation () == theLocalSC; }
new transportAir (provider, client, theParcels, theLocalSC, theRemoteSC);
  { assert theParcels.getLocation () == theRemoteSC; }
new transportRemote (provider, client, theParcels, theRemoteSC, theDestination);
  { assert theParcels.getLocation () == theDestination; }
end sequential;
```

\section{Update constructs}

Like the assert construct described above, the update construct is different from regular protocol constructs in that it is not an interaction pattern. Instead, it is a protocol level construct used to update service objects to make them reflect the state of execution at a certain point of execution of the protocol. There are two forms of the update construct:

\begin{align*}
\text{update.construct} & := \{ \text{'update'} \text{ void.expression} \text{ ';'} \text{ '}' \}\text{'} \\
\text{update.construct} & := \text{'}\text{update}\text{'} \text{ updates 'end' 'update'} \text{ '';'} \\
\text{updates} & := \{ \text{void.expression} \text{ ';'} \text{ +} \\
\end{align*}

The latter form is more readable if the number of updates is large, while the former is better if only a single object must be updated. The void expression(s) specified in the update construct must be expressions operating on a service object, i.e., whose target is the service object, or a component thereof. The service object (or component) must be bound at the time the update is evaluated. The same is true for any arguments used in the expression. The syntax of void expressions is identical to the syntax of Java statement expressions; the Java operators &amp;, ||, and ! may also be written as and, or, and not, respectively. An expression may refer to service objects (section 5.8), service parameters (section 5.9), service indexes (section 5.10), temporary variables (section 5.11), and multiple indexes (section 5.13.9). If the service object is not bound, or if any service objects, service parameters, service indexes, or temporary variables used as arguments in the expression are not bound, an UpdateViolation exception will be thrown by the first task invocation message that is executed after the update was supposed to take place.
Since updates must be tied to a specific point in time at which they should be executed, they can only be specified in ordered sequential constructs, where they can precede and/or follow regular interaction patterns. The updates take place after the preceding task invocation message has completed, but before the following task invocation message will start; the exact time at which they take place is left unspecified.

*Rationale*

The reasons to confine update constructs to sequential interaction patterns is related to what we saw for assert constructs. In this case, value changes are only allowed at certain points in time: the tasks preceding and following the update must all 'see' a constant value of the service object that is updated. If would hardly be useful to state that the service object will be updated during some task that is executed under the same construct. If the enclosing sequential is nested within a concurrent though, there might still be tasks that are executing while the update takes place. This is a confusing situation, that should be avoided.

Note that the updates (just like assertions and task pre- and postconditions) may refer to methods that are defined for objects that are referred to by the contract but not defined as part of that contract. This means that the validity of those methods cannot be established (and hence the validity of the assertion, pre- or postconditions), and that the contract parties have to accept them for what they are worth. It is therefore very important that the classes that are available from the Logistic Contract Space are open-source and read-only, so that every party can verify them before engaging in the execution of the contract.

**Some update examples**

The following update shows how one could mark pallets as unloaded in a drop list:

```java
{ update theDropList.setUnloaded (thePalletTag); }
```

The update operates on a service object, `theDropList`; the pallet tag of the pallet in question is passed as a parameter. The second example is extracted from a contract that describes the unloading of a truck at a platform. The status of the service object is updated to reflect important state changes.

```java
ordered sequential
  // After arrival at the platform, the client has to request unloading
  client : unloadTruck (theTruckID, theActualPlatform);
  // The provider has to unload the truck; it delegates the work to an
  // unspecified (but free!) floormanager.
  { update theDropList.isUnloaded (); }
```

...
After the client has cleared any set-aside pallets from the platform,
the provider has to dismiss the truck.

client : dismissTruck (theTruckID);

The client has to move away from to the given platform

provider : freePlatform (theActualPlatform);

The provider has to stack the pallets; it delegates the work to an
unspecified (but identical) floormanager.

{ update theDropList.isStacked (); }

The pallets in the now stacked drop list are to be unstacked and loaded
on behalf of the distributing clients. Multiple trucks will be involved,
and loading them does need to take place within the same time frame.

{ update theDropList.isLoaded (); }

Some final interactions take place to wrap up the contract as a whole;
this can only happen after all pallets have left the docking station.

end sequential;

The processes that actually cause the changes are represented by comments here. Real contracts will probably formalize the comments by means of process blocks (see section 5.13.12).

5.13.12 Process blocks

As service contracts are used to specify the interactions between a number of logistic actors, they usually don’t specify what happens inside those actors. Under some circumstances though, it might be useful to have some insight into the process logic of the provider or the client. Process blocks can be used for this purpose. A process block is specified as follows:

process_block ::= process_party 'process' interaction_patterns 'end' 'process' ';'
interaction_patterns ::= ( interaction_pattern ';') +
process_party ::= 'provider' | 'client'

The party executing the process is used as a modifier to the process block. Inside the block there is a regular interaction pattern. This interaction pattern is not executed or checked in any way; it just serves as a formalized comment describing what is going on inside the actor. If a task or subcontracted interaction pattern executed from within a process block throws an exception that is not handled locally, it is as if the exception were thrown by the process block itself.

22 Ultimately, the contents of process block might be compiled into the process logic of the specified actors; the current compiler merely ignores process blocks.

23 Exceptions can only be thrown if the tasks and/or sub-contracts inside the process block are really executed, of course.
5.13. The Service Protocol

Rationale

Process blocks are useful when some action within a contract party is performed as a result of an interaction with that party, especially if the party does not execute the action itself, but delegates (part of) the work to one or more subcontractors. In such a case, the process blocks can specify actors and parameters of the delegated service, so the relation between the actor and its delegates becomes clear.

There are some restrictions on the way contract parties can be used within a process block. These restrictions depend on the party that is executing the process block: task invocation messages inside a process block may have that block's executing party as their client, not as their provider. The provider of the task invocation message may be specified as any or as a parameter or variable that was declared to be of an actor type (i.e., a descendent of LincBasicActor). Similarly, subcontracted interaction patterns in a process block may have that block's executing party as their client, not as their provider. The provider of a subcontracted interaction pattern inside a process block may be specified as any or as a parameter or variable that was declared to be of an actor type (again, a descendent of LincBasicActor). See also sections 5.9 and 5.11.

If any is used as an actor in a task invocation message or in a subcontracted interaction pattern, this means that the actor is irrelevant to the current service contract. It does not mean that the actor is irrelevant for the delegating party; only that the appropriate actor is chosen at the process level, and that the service contract is not involved in its selection.

Rationale

The restrictions are meant to make sure that process blocks are used to describe delegation: the block's executing party is a client, and some other actor will play the role of the provider. Whereas any may be used to specify that a certain provider is irrelevant at the process level, it cannot be used directly to specify that several subcontracted services should have the same provider, even though that provider itself is irrelevant. The problem could be solved by using a contract parameter or variable to denote the common provider, and initializing that parameter or variable to any. Since parameters do not allow a default value of any, and variables do not allow a default value at all, this problem is still open.

A process block example

The following protocol fragment shows several services delegated on behalf of the current provider of the service contract. The first two, singlePrepareGetService and singlePreparePutService, are subcontracted from providers that are specified by variables or parameters that were initialized earlier in the contract. The last one, vanService, is subcontracted from an unknown provider. In all cases, the current provider plays the role of the client of the subcontracted service.
ordered sequential
  :
  client : setOrigin (clientOriginBank);
  client : setDestination (clientDestinationBank);
  :
  provider process
    ordered sequential
      unordered concurrent
      new singlePrepareGetService (clientOriginBank, provider);
      new singlePreparePutService (clientDestinationBank, provider);
      end concurrent;
      new vanService (any, provider);
      end sequential;
    end process;
  :
end sequential;

In real contracts, service objects and possibly some service parameters would be passed to the sub-contract instantiations. They were omitted here in order to improve readability of the example.

5.14 The Service Journal

Within the LINC framework, information about the progress made during execution of the contract is made available in the form of a service journal object, which can be manually inspected by means of a journal browser or programmatically queried by the actors themselves and by third parties that might be interested (e.g., customs, taxes, safety boards). The journal tree object will store information about the steps taken in the execution of the contract. The level of detail that can be found in the journal tree depends on what is specified in the journal section of the contract.

In principle every interaction pattern used in the protocol can be logged in the journal. If this is what is specified, the resulting journal will have the form of a tree, like the one shown in figure 5.1. This journal tree contains a node for every executed interaction pattern in the protocol. The root node of the journal always corresponds to the protocol as a whole. The journal section of the contract allows one to specify which individual interaction patterns should be logged in the journal. Logging is enabled by including the keyword corresponding to the construct to be logged into the list following the journal keyword:

\[
\text{service.journal} := \text{'journal'} \text{ journal.specification }; \\
\text{journal.specification} := \text{ journal.construct ( ',' journal.construct ) } * \\
\text{journal.construct} \\
\text{ := 'client' | 'provider' | 'optional' | 'selective' | 'try' | 'timed' | 'protocol' | 'sequential' | 'concurrent' | 'multiple'}
\]
5.14. The Service Journal

![Diagram of a full journal tree]

Each interaction pattern causes specific information to be included in the journal, provided the logging for this construct is enabled. The following list gives the details for every interaction pattern:

**client:** All *task invocation messages* originating from the client will be included into the journal as *client nodes*. The task name will be available as a field of the client node. Any argument values that were passed by the client will also be available as fields of the client node. If the invoked task has a journal value, this value will be inserted into the client node upon successful completion of the task. If the invoked task terminated with an exception, the exception will be inserted in the client node, and the journal value will be left undefined. Client nodes are leaves, they cannot have daughter nodes.

**provider:** All *task invocation messages* originating from the provider will be included into the journal as *provider nodes*. The task name will be available as a field of the provider node. Any argument values that were passed by the provider will also be available as fields of the provider node. If the invoked task has a journal value, this value will be inserted into the provider node upon successful completion of the task. If the invoked task terminated with an exception, the exception will be inserted in the provider node, and the journal value will be left undefined. Provider nodes are leaves, they cannot have daughter nodes.

---

24So the tree presented in figure 5.1 was not fully accurate, since the labels did not denote the node types *client* or *provider*. To improve readability of the journal tree, the invoked task names were included instead of the originators. Since task names may be overloaded between client and server tasks, there is a possible loss of information when omitting the originators.
select: All *selective* interaction patterns will be included in the journal as *select nodes*. Select nodes always have a single daughter node corresponding to the nested interaction pattern that was actually executed.

optional: All *optional* interaction patterns will be included in the journal as *optional nodes*. Optional nodes have no daughter nodes if the nested interaction pattern was skipped, or a single daughter node if the nested interaction pattern was executed.

try: All *monitored* interaction patterns will be included in the journal as *try nodes*. If the interaction pattern that is under the monitored pattern terminated successfully, the try node will have one daughter corresponding to that pattern. If the interaction pattern that is under the monitored interaction pattern terminated with an exception for which the try construct contains a handler, the try node will have two daughters: the left daughter will correspond to the interaction pattern that failed, while the right daughter will correspond to the interaction pattern specified for the exception handler that was executed. In addition, the handled exception itself will be inserted into the try node.

timed: All *timed* interaction patterns will be included in the journal as *timed nodes*. If the interaction pattern that is under the timed pattern terminated successfully, the timed node will have one daughter corresponding to that pattern. If the interaction pattern that is under the timed interaction pattern terminated with an exception that is caused by a violation of one of the timing constraints, the timed node will have two daughters: the left daughter will correspond to the nested interaction pattern, while the right daughter will correspond to the interaction pattern specified for the exception handler that was executed. In addition, the exception itself will be inserted into the timed node.

protocol: All *subcontracted* interaction patterns will be included as *protocol nodes*. These nodes correspond to the protocols of the subcontracted interaction patterns. The provider, the client, and the service objects will inserted into the protocol node. Note that the protocol node corresponding to the top-level contract will always become the root node of the journal tree, even if the protocol is not listed in the journal section. Thus, this keyword only controls the protocol nodes that correspond to subcontracted interaction patterns.

sequential: All *sequential* interaction patterns will be included in the journal as *sequential nodes*. These nodes will have a daughter for each interaction pattern specified in the body of the sequential interaction pattern.

concurrent: All *concurrent* interaction patterns will be included in the journal as *concurrent nodes*. These nodes will have a daughter for each interaction pattern specified in the body of the concurrent interaction pattern. The order in which these daughter nodes occur corresponds to the order in which the nested interaction patterns were executed.
**multiple**: All *multiple* interaction patterns will be included in the journal as *multiple nodes*. These nodes will have a daughter node for every branch, i.e., for every instance of the nested interaction pattern. The order in which these daughter nodes occur corresponds to the order in which the branches were created. If the multiple is indexed, the multiple node will contain a hash table mapping each index value to a daughter node. In addition, if the multiple was closed by one of the multiple constraints, the multiple node will contain information about the constraint that caused the multiple construct to become closed.

Note from the foregoing list that assertions (section 5.13.10), updates (section 5.13.11) and process blocks (section 5.13.12) are not included in the journal tree!

Journal nodes will be created and added to the journal tree as soon as execution of a logged interaction pattern begins; each node will contain a time stamp recording the actual time at which is was created. Depending on the interaction pattern, a node’s contents may be updated during execution. For instance, journal values generated by task invocation messages will be generated and added to the journal tree upon completion of the task. The structure of the journal tree can also change dynamically: new daughter nodes will be added to sequential and concurrent nodes whenever new interactions take place, and the try and timed nodes will change when one of their exception handlers is executed. Note that it is perfectly valid for a task to use information contained in the journal tree as long as it is known that the information is present and stable.

Figure 5.2 shows another journal tree, which now contains a try node with two daughters.

![Diagram](image)

**Figure 5.2**: Full journal tree with exception
The second daughter represents the handling of an exception (by the provider) thrown by the client (from the task `announce ()`). Figure 5.3 shows the journal tree which would have resulted if try nodes were not logged. The actual sequence of interaction patterns is assumed to be identical as the one that resulted in the previous journal tree. Now the try node’s daughters are substituted for the try node itself. Of course it is also possible to log task invocation messages only. In that case the journal tree would be very flat indeed. This would be useful if the client is only interested in the actual sequence of tasks executed and the journal values generated by them. It is important to note that the resulting tree might look different from what one might expect in case the tree contains concurrent nodes dominating sequential nodes. For instance, if a concurrent interaction pattern contains three nested sequential patterns, the resulting full tree would look like figure 5.4. With all sequential and concurrent nodes removed, the tree might look like figure 5.5. The flat journal tree shows the actual order of the messages; this information was lost in the first journal tree because the nodes were inserted into their logical position in the journal tree. Of course, the time stamps included in each node would still reflect the actual time of its
5.15 The Service Report

The service report section has not yet been defined, even though its purpose is well-defined. The purpose is to provide a mechanism for the system to generate a report after the contract has been completed. Since all information about the contract's execution can be obtained from the journal tree, it will be used as input to the service report generator. Most likely, the report generator will be given some tagged representation of the journal tree. The tags will be defined in XML, so the tree can be transformed into any required output, be it log files, billing records, etc. Therefore the report section will probably consist of a set of XML definitions that describe what the output should look like.

```
service_report ::= [ 'report' report_specification ';' ]
report_specification ::= 'Some XML code...';
```

For now, the report section is optional.

5.16 Proposed extensions for Multi-party Contracts

Multi-party contracts are a straightforward generalization of regular 2-party service contracts. Instead of having a single provider and a single client, multi-party contracts usually involve more than two logistic actors. The roles these actors play are not fixed, so the same party could be the provider in one action and the client in another. Additional parties can be specified to form a team of actors or as third parties to which certain tasks can be delegated. Other arrangements are also possible of course.

**Rationale**

In a regular 2-party contract, third parties could be introduced as variables or even as contract parameters. Such parameters or variables could then be used to instantiate subcontracted services inside process blocks, so a simple form of delegation could be specified even in 2-party contracts. The third party would not be a first-class party
in the main contract though, since service variables and service parameters can not
be used as the sender or receiver of a task invocation message. Furthermore, process
blocks are just formalized comments, and the interaction taking place within contained
sub-contracts is verified independently; there is no way to relate actions and exceptional
cases in the sub-contract to what happens in the main contract. Multi-party contracts
offer a solution to both problems: all parties are first class, so all of them can be used
as a sender or receiver of task invocation messages. Also, all parties can be used in
creating sub-contracts; this use is no longer limited to subcontracted services within
process blocks.

Since a multi-party contract describes the interactions between an arbitrary number
of parties, it is possible to view it as a document in which several 2-party contracts
have been merged together. The merging provides extra information though, as the
multi-party protocol provides ordering information that is not present in a collection
of non-related two-party protocols. Although the syntax of the actors section and the
actor tasks section does not suggest a particular hierarchy between contract parties,
this does not mean that hierarchy is necessarily absent: in many cases there will be
actors that take the roles of a client and a main contractor (as in 2-party contracts)
while the remaining actors might act as helpers to either the ‘provider’ or the ‘client’.
This usage is common in case the extra actors have been included to allow clients to
specify which subcontractors should be employed by the provider. E.g., a client might
request the provider of his courier service to use a specific air carrier to transport the
goods, or a client of a car assembly service might tell the provider to use only Rolls
Royce engines (probably because he is more used to buying airplanes). Used this way,
multi-party contracts allow the client to be nosy, by granting him the right to constrain
the possible third parties and subcontractors involved in executing the service.

Another interesting use of multi-party contracts it to include additional actors that
function as a so called patron of a service object, or that correspond to parties that want
to be informed when execution of the contract has reached a certain stage (insurance
agents may come to one’s mind). A patron is a actor that monitors a service object
while it is being handled. It might consist of some hardware (e.g., a transponder chip)
that is attached to the physical service object and that is able to communicate with a
wireless network (or GSM). The transponder might be able to read all kinds of sensors,
like shock, acceleration, or temperature sensors, or even a GPS to track the object’s
position. If any parameter’s value is exceeded, the patron might call for help and
inform other contract parties. If a service object doesn’t show up after being handled,
an exception handler might try to contact its patron in order to find out.

Multi-party contracts are not part of the current LINC language. The syntax of such
contracts is intended as a proposal for implementation in the next version of the language
and the compiler. In order to keep the path to this extension open, the extra keywords
have already been defined as keywords in the initial version of the language (section 5.2).

The specification of multi-party contracts is very similar to the specification of 2-party
service contracts. The overall structure is identical, though the signature section now has
to specify all logistic actors instead of just one provider and one client. Of course there
should also be a tasks section for each logistic actor playing a role in the contract. Finally,
the task invocation messages now need to specify both an origin and an destination of the
message. The protocol structure itself is equivalent to what was specified before. Instead
of repeating the entire specification, we will only present the constructs that are different here.

5.16.1 Multi-party Signatures

The *multi-party signature* is a generalization of the 2-party signature. It names the all logistic actors of the contract as well as their types. The syntax of a multi-party signature is as follows:

```plaintext
multiparty_signature ::= 'signature' multiparty_actor +
multiparty_actor ::= actor_name ':' actor_type '
actor_name ::= simple_actor_name
actor_type ::= name
```

No keywords are used to denote the actor's roles; each actor is specified by an actor name and an actor type. Actor names must be unique within the scope of the contract. Actor types are the same as actor types in 2-party contracts (see section 5.7). It is possible to specify multiple actors of the same type by declaring the actor name as a map type (section 5.4).

**Rationale**

Actor maps are mostly useful for concurrent subcontracting of a service from a number of interchangeable providers: in such a case a concurrent multiple would be used, where the multiple would contain a subcontracted interaction pattern whose provider is the actor map indexed by the multiple index.

**A multi-party signature example**

The following example shows a signature with two logistic actors. The first actor is a `RegionalServiceCenter` that can be referred to as RSC from within the protocol. The second ‘actor’ is actually not a single actor but a map of actors, each of which is an instance of `AirlineCarrier`:

```plaintext
signature
   RSC  : RegionalServiceCenter;
   AC   : map [Positive] of AirlineCarrier;
```

The actual number of airline carriers is not specified as part of the contract. It will be determined when the contract is instantiated. Note an inherent ‘problem’ with maps that are passed as arguments to contracts: there is no way to know which index values correspond to valid (i.e. bound) entries! Most of the time there will be service indexes that can be used to access the map; if not, we will have to rely on external information to pick the proper elements from the map.
5.16.2 Multi-party task sections

Just like a regular service contract has a tasks section for the provider and the client, a multi-party contract has an *actor tasks* section for every actor playing a role in the contract. Each actor tasks section has to refer to an actor name defined in the service signature:

```
  multiparty_task_sections ::= multiparty_task_section +
  multiparty_task_section ::= 'actor' actor_name 'tasks' task_signatures
  actor_name ::= simple_actor_name
  simple_actor_name ::= 'existing actor name'
```

There must be a tasks section for every party defined in the signature section of the contract. The actor name must be the name of an existing actor as defined in the multi-party signature. The task signatures are fully identical to the those in regular service contracts. Refer to section 5.12 for a detailed description of the contents and function of a tasks section.

An example

The example shows a fragment of a multi-party contract that is concerned with transporting parcels by air. The actors are regional service centers (RSC) and airline carriers (AC). As there might be several airline carriers that are used as subcontractors in the transport of parcels, they are best referred to as an actor map. This is of no importance for the task sections though, since all actors in a map are of the same type and are defined by the same actor signature:

```
  actor RSC tasks
    announce (Parcel) throws ParcelTooLarge;
    enter (Parcel) journal ParcelIdentity;
    clear (Parcel) journal ParcelIdentity;
  actor AC tasks
    confirm (Pallet) journal PalletIdentity;
    announce (Pallet) journal PalletIdentity;
    handle (Pallet) journal PalletIdentity;
```

The pre- and postconditions of the tasks are not relevant in the context of this example, so they have been omitted here.

5.16.3 Multiparty task invocation messages

A *multi-party* task invocation message is very similar to a simple task invocation message in a 2-party service contract. The sender and receiver of the message must always be specified though, and actor names must be used instead of the keywords *provider* and *client*. The new rules are as follows:
5.16. Proposed extensions for Multi-party Contracts

```
mparty.simple.interaction_pattern
  ::=  mparty.senderreceiver : task.invocation_message ;
mparty.senderreceiver ::= mparty.sender to mparty.receiver
mparty.sender ::= actor.name [ actor.indexes ]
mparty.receiver ::= actor.name [ actor.indexes ]
actor.name ::= simple.actor.name
simple.it.actor.name ::= 'existing actor name'
actor.indexes ::= '[' actor.index (',' actor.index ) * ']
actor.index
  ::=  service.parameter.reference | service.variable.reference | multiple.index.reference
```

Both the source and the destination of the message must now be given as actor names. These actor names must be defined in the contract signature (section 5.16.1). If an actor is defined as a map, the actor name must be followed by one or more index values. An index value may be a service parameter or a component thereof (section 5.9), a service variable or a component thereof (section 5.11), a service index (section 5.10), or a multiple index (section 5.13.9). The index values must be bound when the protocol flow reaches the interaction pattern, while the referred component of the map must be bound to a valid actor (i.e., a descendant of LincBasicActor). The task invocation message itself is not specific to multi-party contracts. Its description can be found in section 5.13.1.

**Rationale**

As there are no predefined actor roles anymore, the sender and receiver of a message must be referred to by their names. Also, it is no longer possible to infer the sender and the receiver from the partial sender/receiver specification, so both must be explicitly specified. A broadcast to multiple actors can be programmed by nesting the multi-party task invocation message within a multiple construct.

**An example**

In section 5.13.8 an example was presented in which concurrent tasks were employed to construct some car parts to be assembled in a subsequent step. It is not unlikely that the various parts are made by different providers, so all these must be specified in the contract signature:

```
signature
  CF : CarManufacturing.CarFactory;
  CA : CarManufacturing.CarAssembler;
  CM : CarManufacturing.ChassisMaker;
  EM : CarManufacturing.EngineMaker;
  IM : CarManufacturing.InteriorMaker;
```

Ignoring subcontracting for now, we can view the process as a ‘network’ of communicating actors. The actor CarFactory can be viewed as the primary contractor, while the
CarAssembler and the various component manufacturers are secondary actors in this contract. The following protocol starts with the CarFactory sending an assembleCar () message to the CarAssembler. The latter then sends messages to the various manufacturers who can perform their tasks in parallel. After all tasks have been completed, the CarAssembler reports back to the CarFactory, which then issues a message to test the assembled car:

```
protocol  
ordered sequential  
  CF to CA : assembleCar ();  
unordered concurrent  
  CA to CM : constructChassis ();  
  CA to EM : constructEngine ();  
  CA to IM : constructInterior ();  
end concurrent;  
  CA to CF : completedCar ();  
  CF to CA : testCar ();  
end sequential;
```

The above example shows how multi-party contracts can be used to specify interactions that are related to each other or that might have timing constraints with respect to each other, even though those interactions are not between a single client-provider pair. No subcontracting is used, so the contract suggests there is no hierarchy between the actors.

### 5.16.4 Multiparty subcontracted interaction patterns

The changes to subcontracted interaction patterns are similar to what we have seen above: all references to the provider and the client of the current contract have been replaced by references to the actor names. In addition, the sub-contract may itself be a multi-party contract and thus have more than two actor arguments:

```
multiparty_subcontracted_interaction_pattern  
::= 'new' service_name (' multiparty_service.arguments ') ' ; '  
service_name ::= 'existing service name'  
multiparty_service.arguments  
::= multiparty_service.actors  
  [' ; service.objects '] [' ; service.parameters '] [' ; serviceindexes ]  
multiparty_service.actors ::= multiparty_service.actor (' ; multiparty_service.actor ) *  
multiparty_service.actor  
::= actor_name | service.parameter_reference | service_variable_reference  
actor_name ::= simple.actor.name  
simple.it actor_name ::= 'existing actor name'
```

All other aspects of a multi-party subcontracted interaction pattern are identical to the regular subcontracted interaction pattern.
An example

The following example is similar to the example in section 5.16.3 but uses sub-contracts:

```
protocol
  ordered sequential
    CF to CA    : assembleCar ();
  unordered concurrent
    new constructChassis (CM, CA);
    new constructEngine (EM, CA);
    new constructInterior (IM, CA);
  end concurrent;
  CA to CF    : completedCar ();
  CF to CA    : testCar ();
end sequential;
```

The three construction steps have now been ‘upgraded’ from tasks to separate sub-contracts; the contract now reflects the fact that some of the parties are specified by the party that instantiates the contract. There is a natural hierarchy: the carFactory is the client, while the CarAssembler is the provider of the contract.

5.16.5 Multiparty exception handlers

The final difference with 2-party contracts applies both to monitored interaction patterns and to timed interaction patterns. Both constructs may contain informal exception handlers (see section 5.13.5) that can refer to actors of the contract. The new rules regarding the handler invocation are as follows:

```
multiparty_handler_invocation
  ::= 'to' multiparty_service_actors ':' handler_invocation_message
multiparty_service_actors ::= all | multiparty_actor_list
multiparty_service.actor.list ::= multiparty_service.actor ( ',', multiparty_service.actor ) *
```

Multiparty service actors have been defined above. Of course there are minor changes in the rules for monitored and timed interaction patterns too, but these are straightforward and will not be described here.
Chapter 5. The LinC Language Specification
Part III

Case Studies
The third part of this thesis presents two detailed case studies in which the LINC language is used to describe the interactions between a number of logistic actors. The first case study is concerned with the transportation of valuable goods between two banks on behalf of a customer who is holding a safe deposit box at either bank (chapter 6). The second case study is larger in scale: it describes a complete cross docking station where trucks deliver goods on behalf of some producers, while other trucks pick up the goods and redistribute them to resellers (chapter 7). The system described by the latter case study is currently in the process of being implemented.

The case studies described here are intended as an empirical validation of the language; their purpose is to show that systems of this kind can be effectively described by LINC contracts. Apart from being meant to show that the language is descriptive, i.e., usable as a specification language, compiling and using the contracts to verify a working system also shows that the language allows for a formal verification of the actual communication taking place. The case studies are not intended as a formal validation of the language; such a validation would only be possible after its semantics has been formally defined. In part V we will however argue that the language is adequate, i.e., that it can be used to describe the communication patterns between any pair of Java objects in sufficient detail, and that the corresponding runtime system can be used to assign a suitable semantic structure to a dialogue that is taking place between actors.

Finally, the case studies are meant to show prospective users how the language might be used; in some cases different solutions will be compared to show the strong and weak points of the various language constructs. To some readers, the sheer amount of contracts and the apparent complexity of some of them may seem like overkill for the design of logistic systems. We do not share this view, of course: one of the main reasons to design the LINC language was to allow logistic engineers to specify the many details, thereby allowing the logistic process to be guarded during its execution. Of course there will be room for discussion as to where to draw the line that separates what is put into the contract and what is left to the process logic. Many such discussions have in fact taken place while the case studies were in progress. A related note about the amount and complexity of the contracts presented in this part is concerned with the reader’s patience; at some point, most readers will feel that what is going to follow is just more of the same thing. This is true, of course, but only partially, as every contract presented here has led to new insights, and sometimes to changes in the language. It is for this reason that the case studies are published in their entirety. Readers are of course free to skip as many contracts as they like, only coming back when more detailed information is needed.
Chapter 6

Case Study: valueTransferService

The first case study is concerned with the transportation of valuable goods from one bank to another on behalf of a customer who is a client of both banks and who holds a safe deposit box at either one of them. This customer asks some transportation company to arrange a transport for him. The transportation company must be recognized as a trusted party by both banks, and must have (probably long-term) contracts with the banks involved to pickup or deliver valuables.

The logistic actors involved in this study are the owner of the goods, the bank where these goods are currently stored, the bank where they will be stored in the future, a transportation company that is charged with coordinating the entire transport, and a company providing armoured car services for the purpose of executing the physical transportation tasks. All these actors are shown in figure 6.1; ING and ABN are bank names, while Transporter and Van are the ‘logical’ and the ‘physical’ transporter, respectively. The actor referred to as Client is the bank customer that makes the transportation request.

![Diagram of the value transport service](image)

Figure 6.1: Diagram of the value transport service

The arrows in this figure represent 2-party service contracts; they are directed from the
client to the provider and labeled with the service contract name\(^1\). Thus, the figure shows 8 different service contract instances, two of which are distinct instances of the same model contract (i.e., \texttt{safeRental}). The lifetimes of the various contracts can be quite different: some are like \textit{one shots} concerned with a specific task of finite duration, while others are like long term generic contracts possibly involving many subcontracted services to be executed under their control. The physical transport related contracts are good examples of the first kind, while the more permanent relations between parties (like \texttt{safeRental}) are examples of the second kind of contract.

Although all mentioned contracts will be part of the Logistic Contract Space, only some of them may be useful c.q. visible and/or accessible to the average customer, i.e.: \texttt{safeRental} and \texttt{valueTransferService}. The remaining contracts are intended to be executed by specialized actors, whose rights to view and access the Logistic Contract Space may be less restricted. As we will see in section 9.1, the current framework does not have any provisions to distinguish between users or to enforce visibility and/or access restrictions. Practical implementations of the Logistic Contract Space are however free to devise a specialized mechanism or to rely on the underlying system's mechanisms to enforce such visibility/access restrictions.

### 6.1 The service actors

Just like the service contracts themselves, most logistic actors for the value transport service will be part of the Logistic Contract Space. This is the only option for untrusted parties, which can only participate in a contract by downloading existing code (which probably takes the form of an \textit{applet} running in a web-browser). Trusted parties may either upload their actor code or keep it at their local machines; in any case the actor code has to adhere to the standards prescribed by the LINC platform (see chapter 9). Before turning our attention to the service contracts themselves, it is useful to take a closer look at the various actors of figure 6.1:

**Client:** The client of the value transport service is assumed to be a regular bank customer, i.e., an individual who owns a bank account and who has rented a safe deposit box at one of the bank's offices (in fact, this customer has to have rented \textit{two} safes at different banks, or at least at two different branch offices of the same bank). As this customer is a party in the outside world, it must be represented by a software entity which is the actor in the software sense. Given the LINC platform, actors like these can easily be implemented by Java applets and executed in a web-browser.

A possible implementation of the value transport service consists of a private webpage that is dedicated to the transport service. The main contents of this page is

\(^1\)Multiparty contracts can be visualized in a similar way, but since more than 2 parties are involved we would draw a straight line or arrow between any pair of parties that communicates. If a particular pair of actors from the set has a clear client-provider relation, an arrow would be used; if not, a regular line would be drawn. All lines/arrows would be labeled with the same contract name.
generated by an applet implementing the client’s graphical user interface. If the web page may only be accessed by bank customers after they have identified themselves to their bank, the transport service does not have to bother anymore about the client’s identity. Since two banks might play a role in the contract, the client may still have to identify himself to the second bank.

**Transporter:** In contrast to what the name might suggest, the transporter is a controlling or mediating actor rather than an actor that performs any physical transports itself. It might be implemented as a distinct party or as a distinct role of one of the other parties. When a distinct party is used, it will probably take the form of a common organization that is established by a consortium of banks. When implemented by an existing actor, it must be a distinct role of either a bank or the provider of the vanService; this would be a good example of a single party engaging in more than one service contract.

**ABN, ING:** Compared to daily banking processes, the services offered by the banks in this case study are rather minor. Nevertheless, the corresponding actors may rely on regular banking processes like customer identity verification; the actors form an interface to a small (but important) part of the regular administrative software operated by the banks.

**Van:** The provider of the vanService corresponds to a party owning a fleet of armoured cars. It is the party that has to pick up the goods at one bank and to deliver them at the other. As such, it must be a party that is known to and trusted by the banks involved. Trustworthiness is guaranteed by asking the Transporter to identify the Van when it requests the banks to prepare the transport. If the Van were not trusted, these preparations would simply fail.

Obviously, the actors ABN and ING are both instances of a single actor class (which might be known as Bank). Whereas such institutional actors may be allowed to keep the code of their actors private and only known to themselves (obviously, they still would have to implement the interfaces that are prescribed by the service contracts they are involved in), regular bank clients will probably not be allowed to use their own implementation of the Client actor. This client actor, which is involved in just two service contracts (safeRental and valueTransferService) will probably be defined as an applet and stored in the Logistic Contract Space to be downloaded into the customer’s web-browser.

### 6.2 The service contracts

This section presents the service contracts between the various parties. These contracts are detailed enough to be used as the basis of an implementation. Some of them will

---

2If banks were to adopt a centralized identification system, it would be sufficient to identify the client to one bank only.
be publicly accessible, while others may belong to a limited-access banking domain in the Logistic Contract Space. Four contracts are long-lived: two instances of safeRental, as well as multiplePrepareGetService and multiplePreparePutService. The other contracts (valueTransferService, singlePrepareGetService, singlePreparePutService, vanService, getService, and putService) are instantiated for the job at hand and will cease to exist when the transport has been successfully completed.

6.2.1 valueTransferService

The valueTransferService contract presented in figures 6.2 and 6.3 describes the interaction between a client who wants to have some valuables transported from one bank to another, and a provider who offers a coordination service to accomplish this. The main provider of this coordination service must be a Transporter. The client, which may be any party that owns a safe deposit box at either bank, approaches the provider to have the entire transport done and expects everything to be arranged for him. The Transporter relies on a subcontracted service, i.e., vanService, to have the physical transport done. As the Transporter needs to make the necessary preparations at the banks, it can only arrange transports between banks with which it has the following (long-term) contracts: prepareGetService with the origin of the transport, and preparePutService with the destination of the transport.

Notes to figures 6.2 and 6.3:

- The provider of the valueTransferService must be a Transporter, while the client can be any client that owns a safe. Although we would like to express the fact that the client has to have active safeRental contracts with the two banks involved in the transfer, there is no way to explicitly state this fact. However, the client is specified as a SafeOwner, which is the same actor as the client of the safeRental contract described in section 6.2.2. Though not formally implied, the existence of such contract instances is at least suggested by this choice of actor. Being the client of the safeRental service too, the SafeOwner needs to implement the client tasks of the current service as well as those of the safeRental contract (see section 6.2.2).

- The service object of the contract is the parcel that is to be transported. Since a default value is specified, the parcel does not need to be known at contract instantiation; it may be passed at the time the transport is requested by requestTransport (). If a parcel was specified at contract instantiation, it must be identical to the parcel passed in the requestTransport () message.

- There is only one provider task, i.e., requestTransport (). This task has a precondition stating that the parcel must be in a bank safe at the origin bank before the transport can be requested. Its postcondition guarantees that the parcel will be in a bank safe at the destination bank after the task terminated successfully. The service object is used as an argument, as well as the origin and the destination banks of the
6.2. The service contracts

description
  domain : ValueTransport;
  service : valueTransferService;

actors
  provider : Transporter;
  client : SafeOwner;

objects
  theParcel : Parcel := null;

provider tasks
  requestTransport
    (theParcel : Parcel,
     theOrigBank : Bank,
     theOrigSignature : Signature,
     theOrigKey : Key,
     theDestBank : Bank,
     theDestSignature : Signature,
     theDestKey : Key)
  pre theOrigBank.getSafe (theOrigSignature).contains (theParcel)
       and not theDestBank.getSafe (theDestSignature).contains (theParcel)
  post theDestBank.getSafe (theDestSignature).contains (theParcel)
       and not theOrigBank.getSafe (theDestSignature).contains (theParcel)
  throws InvalidOriginBank,
       InvalidDestinationBank,
       InvalidOriginSignature,
       InvalidDestinationSignature,
       InvalidOriginKey,
       InvalidDestinationKey,
       TransportDenied,
       DeliveryProblem

  journal TransportRecord;

client tasks
  unsupportedBank (Bank);
  invalidSignature (Signature);
  invalidKey (Key);
  reportDenial (DeniedDetails);
  reportProblem (ProblemDetails);

variables
  clientOrigBank : Bank;
  clientDestBank : Bank;
  clientOrigSignature : Signature;
  clientDestSignature : Signature;
  clientOrigKey : Key;
  clientDestKey : Key;
  denialDetails : DeniedDetails;
  problemDetails : ProblemDetails;

Figure 6.2: valueTransferService (part 1 of 2)
protocol
  try
    nested concurrent
      // If this task completes without exception, we can be sure everything is OK
      client : requestTransport
        (theParcel, clientOrigBank, clientOrigSignature, clientOrigKey,
         clientDestBank, clientDestSignature, clientDestKey);
    // 'Implementation' of the provider task
    provider process
      try
        ordered sequential
          unordered concurrent
            new singlePrepareGetService
              (clientOrigBank, provider, theParcel, theParcel);
            new singlePreparePutService
              (clientDestBank, provider, theParcel, theParcel);
          end concurrent;
        // Assume that the preparations were successful
        new vanService (any, provider, theParcel);
      end sequential;
      when ProcessError :
        // Throw appropriate exception to the client
      end try;
    end process;
  end concurrent;
  when InvalidOriginBank :
    to client : unsupportedBank (clientOrigBank); reject;
  when InvalidDestinationBank :
    to client : unsupportedBank (clientDestBank); reject;
  when InvalidOriginSignature :
    to client : invalidSignature (clientOrigSignature); reject;
  when InvalidDestinationSignature :
    to client : invalidSignature (clientDestSignature); reject;
  when InvalidOriginKey :
    to client : invalidKey (clientOrigKey); reject;
  when InvalidDestinationKey :
    to client : invalidKey (clientDestKey); reject;
  when TransportDenied :
    provider : reportDenial (denialDetails); reject;
  when DeliveryProblem :
    provider : reportProblem (problemDetails); reject;
end try;

Figure 6.3: valueTransferService (part 2 of 2)
6.2. The service contracts

transport, both client signatures, and both keys that were given to the client when it announced the transport as part of its safeRental contract.

- The client tasks are: reportDenial (), reportProblem (), unsupportedBank (), invalidSignature (), and invalidKey (). They are all exception handling tasks used to inform the client of the various possible problems concerning the transport.

- The expressions in the pre- and postcondition of requestTransport () can only be fully evaluated if getSafe () returns a valid safe. If not, the subsequent contains () would throw a NullPointerException. Since any exceptions thrown during evaluation of conditions and/or assertions cause the condition or assertion to evaluate to false, this does not cause a problem.

- The protocol is quite simple: it is just a single invocation of the provider task that is enclosed within a try construct to handle possible errors and an nested concurrent construct to be able to provide more details about task’s execution (see below).

- The exception InvalidOriginBank or InvalidDestinationBank will be thrown if the source or the destination bank are not supported by the Transporter. This could happen if the provider is not active in the region of one or both banks, or if it has no contract with one of the specified banks. An informal client exception handler task (unsupportedBank ()) is used to inform the client; the bank that is causing the problem is passed as a parameter to this handler task. Since there is no way to complete the contract, both exception handlers end with a reject.

- Similarly, the exceptions InvalidOriginSignature or InvalidDestinationSignature will be thrown if one or both client signatures are not recognized by the banks, and the exceptions InvalidOriginKey or InvalidDestinationKey will be thrown if the client did not provide the proper keys for this particular transport (the keys can be obtained under the safeRental contracts with both banks; they are specific for the current provider). The handling of these exceptions is the same as before: an informal client exception handler task will be called upon, passing the offending datum as an argument, after which the entire contract will be aborted.

- The exception TransportDenied will be thrown if for some reason one of the banks cannot honour the requested transport. This could happen if there is some administrative problem with the client or with the transport. The last exception is DeliveryProblem; this exception might be thrown if the goods didn’t make it to the destination. Because the information to be passed to the client cannot be deduced from the contract state (only the provider can tell what exactly went wrong), formal exception handlers containing regular task invocation messages are used instead of informal exception handler calls of the form ‘to client’. As before, the contract will be terminated.

- Note that in all of these cases the client will also be notified of the exception, since the method call in its business logic will receive the exception in the usual way.
The monitored interaction pattern is used to specify how the protocol should change its flow as a result of the exception. Even if the flow is not changed, a monitored interaction pattern should be included to make this fact absolutely clear to both contract parties. For the same reasons, a contract is not allowed to end with an exception (this does not apply to the ProcessError and TimeoutError exceptions), so each named exception that might be thrown by a task must have a handler for it in the protocol.

- The main task invocation message is followed by a provider process block detailing the most important aspects of the implementation of the provider task. Since the process block represents the implementation of the preceding task, the task invocation message and the process block need to be enclosed in a nested concurrent construct for the whole to work. This is especially true for the postconditions on the task signature, which obviously refer to the situation that exists after the task, and hence the process block, have terminated.

- Usually, a process block that is used to clarify the implementation of a preceding task invocation would contain assertions that are equivalent to the pre- and postconditions on the invoked task. In this case it is not possible to do that, because it is impossible to rewrite the pre- and postconditions to expressions that have the service object as their target: the service object of the current contract, a Parcel, is not aware of its own location.

- The provider process shows that the transporter has to prepare the transport with both involved banks, and how it sub-contracts the actual transport from an arbitrary provider of vanService. The preparations for the transport can occur concurrently; they have to be completed before execution of the subcontracted vanService can start. The current provider is the client of all sub-contracts. The provider of the preparation services are the banks, while the selection of a provider for the vanService is up to the provider.

- Note that the vanService contract can only be executed if both prepare services resulted in the respective providers to confirm the transport and to supply a temporary key. If the provider of the prepareGetService should deny the requestGet (), or if the provider of the preparePutService should deny the requestPut (), the subsequent vanService must not be executed. Since it is not possible to terminate a sub-contract by throwing a named exception, this can only be accomplished by making the failing service throw a ProcessError exception. Since we wouldn't want this exception to terminate the entire contract, it should be caught inside the process block itself.

- One cannot see from this contract that execution of the singlePrepareGetService and the singlePreparePutService subcontracted services requires the pair of keys that was passed by the current client as part of the requestTransport () invocation, nor that it results in the current provider acquiring a another pair of keys that is needed to execute the vanService. While the former set of keys could be passed to
the subcontracted service as parameters, the newly obtained keys would need to be returned from those sub-contracts, which is not possible in the current language. As these are implementation details anyway, it is not necessary to include them in the contract.

- The parcel is passed as a service object to all subcontracted services. In the case of `singlePrepareGetService` and `singlePreparePutService`, the parcel must passed a second time, i.e., as a multiple index. This is necessary because the contracts for these services contain a service index corresponding to the parcel. Note that the service object itself is never updated in this contract; it is assumed that the parcel is not aware of its own location.

- During the course of the transport, different parties become responsible for the parcel's well-being: first the origin bank, followed by the transporter (who delegates its responsibility to the provider of the `vanService`), then the van, and finally the destination bank. This transfer of responsibilities is reflected by the passing of the parcel as a service object to the various contract instantiations. By accepting a service object in the instantiation of a service contract, the actors agree to share responsibility; passing the service object as an argument in a task invocation message transfers responsibility to the actor that provides that task. Similarly, if the service object is passed to a subcontracted service, responsibility is transferred to the actors of that subcontracted service. In that case, the contracting hierarchy reflects a chain of responsibilities instead of a single responsibility transfer.

### 6.2.2 safeRental

In order to be able to request a transport between two different bank safes, a client must have rented a safe deposit box at both banks, i.e., there must be an active `safeRental` contract with either bank at the time of the transport. As soon as the rental period starts, the client will get a personal electronic signature from the bank; this signature (which is only valid as long as the contract is active) can be used to gain access to the safe. Clients may request access to their safes; when doing so, they must identify the party that is going to do the access (this may or may not be the client itself). When access is granted, the bank provides the client with an electronic key that can be used to access the safe exactly once. This key will expire after a certain amount of time, and can only be used by the specified party. A key can only be used as long as the contract is active. The service contract is shown in figure 6.4. Notes to the figure:

- Since the client has to perform some tasks in this contract, it is specified as a special party, i.e., a `SafeOwner`. To stress the fact that the client of the `valueTransferService` has to own at least one safe, that client was specified as being a `SafeOwner` too. As a result, this actor type has to implement both the tasks that are required by the current service contract and those that are required by the `valueTransferService` contract.
description
  domain : ValueTransport;
  service : safeRental;
actors
  provider : Bank;
  client : SafeOwner;
objects
  theSafe : Safe := null;
provider tasks
  assignSafe ();
  requestKey (theRequestID : RequestID, theSafe : Safe, theSignature : Signature,
              theActor : LincBasicActor);
  freeSafe (theSafe : Safe, theSignature : Signature) pre theSafe.isEmpty ();
client tasks
  useSafe (theSafe : Safe, theSignature : Signature);
  useKey (theRequestID : RequestID, theKey : Key);
  discardSafe (theSafe : Safe);
  reclaimedSafe (theSafe : Safe);
variables
  theSafe : SafeID;
  theSignature : Signature;
  theActors : map [RequestID] of LincBasicActor;
  theKeys : map [RequestID] of Key;
protocol
  ordered sequential
    client : assignSafe ();
    provider : useSafe (theSafe, theSignature);
    indexed (theRequestID : RequestID)
    sequential multiple ()
      client : requestKey (theRequestID, theSafe, theSignaature, theActors[@theRequestID]);
      provider : useKey (theRequestID, theKeys[@theRequestID]);
    client process
      // The client can now use the key to access the safe once
    end process;
  end multiple;
select
  timed (duration 30 Days)
    ordered sequential
      provider : discardSafe (theSafe);
      client : freeSafe (theSafe, theSignature);
    end sequential;
  when DurationError :
    to client : reclaimedSafe (theSafe); reject;
    end timed;
    client : freeSafe (theSafe, theSignature);
  end select;
end sequential;

Figure 6.4: safeRental
• The safe is used as a service object here. The initial value will be null; a proper
value will be assigned as soon as the provider assigned a safe to the client following
the latter’s request.

• The protocol starts with the client requesting a safe from the bank. The bank then
returns a safe id and an electronic signature to be used in client requests concerning
the safe. After that, the client may use the safe for its own purposes. Safes are rented
for an indefinite period; real-world contracts might require the client to specify a
rental period.

• The client can access the safe at any time, but only after it has issued a formal request
to do so. Two kinds of access are defined: to get something from the safe, and to
put something into the safe. Each request is made in two stages: first, the client has
to ask the bank to make preparations for then access; second, he has to request the
access itself. The two stages of a request may or may not be executed by different
parties.

• Since the client may sub-contract a third party to arrange the entire transport, it
has to announce that party to the bank when requesting access. The client does not
have to tell the bank if the access is needed to ‘get’ or to ‘put’ something: it just
sends a requestKey() task invocation message to the bank. The provider will then
honour the request by invoking the client’s useKey() task, supplying the electronic
key needed to make further requests to the bank. This key will be specific to the
third party that was mentioned by the client; any other party trying to execute a
prepareGet or preparePut contract will be rejected by the bank, even if it got hold
of a key. If the safe would always be accessed by the client himself, we could have
specified client for the actor; in that case, the client actor would have to be a
Transporter as well.

• A comment indicates that the client is taking unspecified actions to access the safe. We
could have included a subcontracted service (inside a process block) here to perform
the safe access on behalf of the client. The provider of the subcontracted service
would be specified as theActors[@theRequestID], and the client as client. The
key (i.e., theKeys[@theRequestID]) would need to be passed as a service parameter
to the subcontracted service.

• An indefinite multiple is used to specify that the client can access the safe as many
times as necessary. As in any multiple, a multiple index (theRequestID) is required to
relate the tasks being executed to each other. Since the returned key will be different
for every instance of the multiple body, it is made a map variable, indexed by the
multiple index. The same holds for the actor to which the job of accessing the safe is
delegated.

• The safe and the signature are the same for any instance of the multiple’s nested
interaction pattern. Since it is more common inside multiples for task arguments to
be different for each instance, the compiler may issue a warning for each argument
that is not a multiple index or a map variable indexed by a multiple index. In cases like this, such warnings can be safely ignored.

- The bank may terminate the contract at any time, by sending a discardSafe () request to the client. The client may continue to use the safe until it has issued a freeSafe () request; it must do so within 30 days though. If it failed to do so in time, a DurationError will be thrown when it finally does; the client exception handler task reclaimedSafe () will be called to inform the client of the possibly disastrous consequences. There is no need to send a similar message to the provider, since the latter will already know that the client is (too) late. The handler ends with a reject clause to make it absolutely clear that the contract will be aborted.

- The client may also take the initiative itself to terminate the safe rental by informing the bank that it does not need the safe anymore.

6.2.3 multiplePrepareGetService, multiplePreparePutService

Any Transporter that wants to have goods collected from or delivered to a bank needs to have an active multiplePrepareGetService contract and a multiplePreparePutService contract with these banks. These long-term contracts merely state that the transporter intends to establish a number of instances of the singlePrepareGetService and the singlePreparePutService contracts with the respective providers\(^3\). The preparations for actual pickup or deposit of goods on behalf of some safe owner are performed as a part of the latter two contracts. Since both contracts are almost exact copies of each other, only one of these services will be described below (see figure 6.5).

Notes to the figure:

- There is no service object at this level, and there are no provider or client tasks. This is because the contract is only used to specify that a number of lower level contracts will be established to prepare individual transports.

- A parameter section (containing one parameter) is used here to pass the duration of the contract (expressed as a number of months). The parameter value is used in the duration clause of the multiple.

- Within the multiple interaction pattern, the singlePrepareGetService service is instantiated for every individual parcel to be transported, so the parcel is used as a multiple index. The parcel is also passed as a service object of the newly instantiated subcontract. Since it is the multiple index, it must even be passed a second time to the subcontracted service, this time as a service index!

\(^3\)Note that in figure 6.1 the 'multiple' and 'single' variants of the contract are taken together under the label prepareGetService.
6.2. The service contracts

description
    domain : ValueTransport;
    service : multiplePrepareGetService;
actors
    provider : Bank;
    client : Transporter;
objects
    none;
parameters
    theNumberOfMonths : Natural;
provider tasks
    none;
client tasks
    none;
protocol
    indexed (theParcel : Parcel)
    concurrent multiple (duration : theNumberOfMonths)
        ordered sequential
            new singlePrepareGetService (provider, client, @theParcel, @theParcel);
        end sequential;
    end multiple;
journal
    client, provider, select, try, sequential, concurrent, multiple;

Figure 6.5: multiplePrepareGetService
• This contract describes the `multiplePrepareGetService` contract. The corresponding `multiplePreparePutService` can be obtained from the contract shown by replacing the name of the contract and by substituting `singlePreparePutService` for `singlePrepareGetService` in the multiple body.

6.2.4 singlePrepareGetService, singlePreparePutService

To arrange individual parcel transports, transporters have to instantiate the services `singlePrepareGetService` and `singlePreparePutService` with the banks involved. Both contracts are instantiated as a subcontract of the ‘multiple’ variants of the same contracts (described in the previous section). The contracts specify the interactions between the client (a `Transporter`) and the provider (a `Bank`) to prepare the bank for pickup or delivery of a single parcel. A request to pickup or deliver something must always be made on behalf of a client of the bank; each such request has to pass the digital signature of that client as well as the key that was obtained by that client under the `safeRental` contract. If the preparation requests are honored by the respective providers, the latter will return a pickup and a delivery key to the `Transporter`, who can then forward them to the physical transporter (i.e., the provider of the `vanService`) to get the goods from a bank and to deliver them. Again, both contracts are almost exact duplicates of each other, so only one of them will be shown here (see figure 6.6). Notes to the figure:

• The indexes section shows that the service can be subcontracted from inside a multiple construct. This has important consequences for all task invocation messages that are exchanged under control of the contract: they must all include a reference to the multiple index, and most non-index arguments will be map variables. Future implementations might be able to drop the requirement of passing multiple indexes to subcontracted services, thereby moving this burden from the contract writer to the implementation.

• When the bank receives a `requestGet()` message, it is supposed to prepare the specified goods (the `Parcel`) for transport and have them available when they are collected. The request has to include the parcel, the safe, the original client’s signature, and the key that was given to this client when it first announced the transport to the bank. For safety reasons, this key is specific to the party which is executing the current contract, i.e. the current provider. Note that the safe and the client signature are the same for each instance of the contract, so these variables need not be maps.

• Since most of the time the goods will be collected by a third party (e.g. an armoured car), the same mechanism as before is used when requesting the actual transport: the current provider has to specify the actor which will collect the goods, and will be given a key that is specific for that actor. So the bank responds to the request with a `confirmGet()` message supplying the new key. Alternatively, the bank is allowed to send a `denyRequest()` message to the transporter.
6.2. The service contracts

description
domain : ValueTransport;
service : singlePrepareGetService;

actors
provider : Bank;
client : Transporter;

objects
theParcel : Parcel;

indexes
theParcel : Parcel;

provider tasks
requestGet
(theParcel : Parcel,
theSafe : Safe,
theSignature : Signature,
theKey : Key)
pre provider.owns (theSafe) and theSafe.contains (theParcel)
throws InvalidSignature, InvalidKey;

client tasks
confirmGet (theParcel : Parcel, theKey : Key)
denyGet (theParcel : Parcel, theDetails : DenialDetails)

variables
clientSafe : Safe;
clientSignature : Signature;
clientOldKey : map [Parcel] of Key;
clientNewKey : map [Parcel] of Key;
denialDetails : map [Parcel] of DenialDetails;

protocol
try
ordered sequential
client : requestGet (@theParcel, clientSafe, clientSignature, clientOldKey[@theParcel]);
select
provider : confirmGet (@theParcel, clientNewKey[@theParcel]);
provider : denyGet (@theParcel, denialDetails[@theParcel]);
end select;
end sequential;
when InvalidSignature :
reject;
when InvalidKey :
reject;
end try;

journal
client, provider, select, try, sequential, concurrent, multiple;

Figure 6.6: singlePrepareGetService
• A precondition has been specified for requestGet (): the bank must own the given safe, and that safe must indeed contain the parcel to be transferred. It is the client's responsibility to make sure that these conditions are satisfied. Note that the validity of the signature and the key are not enforced by a precondition; if either of these is not valid, one of the exceptions InvalidSignature or InvalidKey will be thrown.

• If one of the exceptions is thrown, the contract will be terminated immediately. The exceptions speak for themselves, so there is no need to call a client exception handler task.

• There might be additional requirements related to the request, e.g., the requirement that a parcel can only be prepared for transportation once. The current set of contracts specifies a single transport, and can only offer guarantees with respect to that transport when it is assumed that no other parties are manipulating the same parcel at the same time. This in fact applies to the service objects of any service contract! Exclusive use conditions like this are difficult to express as preconditions, and would either require that service objects are protected from concurrent use by more than one contract instance (sub-contracts should be allowed, though), or require that the semantics of preconditions were changed to exclude concurrent use on a per-task basis.

• The singlePreparePutService contract exactly mirrors the singlePrepareGetService. The protocol is identical, except that all occurrences of 'Get' in the contract have to be replaced by 'Put'. Obviously, the precondition on preparePut () has to be different from prepareGet ()'s precondition: preparePut () instead requires that the parcel is not already at the destination. Note that the singlePrepareGetService and singlePreparePutService contracts can easily be combined into a single contract.

6.2.5 vanService

The physical transport of parcels is specified by the vanService contract. The vanService service consists of driving an armoured car to the origin bank and picking up the valuables, then transporting them to the destination bank and having them delivered there. In order to execute these steps, the provider of the vanService has to create one-shot contracts with both banks to have the getService and the putService performed. To prove that it is entitled to collect and deposit the goods on behalf of the client and to make sure that the preparations were successfully completed at both sides, the provider of vanService has to pass the keys that were given to it by the Transporter to the providers of these services, i.e., the origin and destination Banks. During the course of the getService, the origin bank will get a receipt from the Van, and during the course of the putService, it will get a receipt from the destination bank. If all contracts terminate normally, these receipts are never used. If something went wrong or if the initiating client requests for them, the Transporter will ask the origin Bank as well as the Van to present their receipts. The handling of these receipts belongs to the process logic of the provider of the vanService, and is not shown.
6.2. The service contracts

<table>
<thead>
<tr>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>domain</td>
</tr>
<tr>
<td>service</td>
</tr>
<tr>
<td>actors</td>
</tr>
<tr>
<td>provider</td>
</tr>
<tr>
<td>client</td>
</tr>
<tr>
<td>objects</td>
</tr>
<tr>
<td>theParcel</td>
</tr>
<tr>
<td>provider tasks</td>
</tr>
<tr>
<td>executeTransport</td>
</tr>
<tr>
<td>(theParcel</td>
</tr>
<tr>
<td>theOrigBank</td>
</tr>
<tr>
<td>theOrigSignature</td>
</tr>
<tr>
<td>theOrigKey</td>
</tr>
<tr>
<td>theDestBank</td>
</tr>
<tr>
<td>theDestSignature</td>
</tr>
<tr>
<td>theDestKey</td>
</tr>
<tr>
<td>pre</td>
</tr>
<tr>
<td>theOrigBank.getSafe (theOrigSignature).contains (theParcel)</td>
</tr>
<tr>
<td>and not theDestBank.getSafe (theOrigSignature).contains (theParcel)</td>
</tr>
<tr>
<td>post</td>
</tr>
<tr>
<td>theDestBank.getSafe (theDestSignature).contains (theParcel)</td>
</tr>
<tr>
<td>and not theOrigBank.getSafe (theDestSignature).contains (theParcel)</td>
</tr>
<tr>
<td>throws</td>
</tr>
<tr>
<td>InvalidOriginBank,</td>
</tr>
<tr>
<td>InvalidDestinationBank,</td>
</tr>
<tr>
<td>InvalidOriginSignature,</td>
</tr>
<tr>
<td>InvalidDestinationSignature,</td>
</tr>
<tr>
<td>InvalidOriginKey,</td>
</tr>
<tr>
<td>InvalidDestinationKey,</td>
</tr>
<tr>
<td>DeliveryProblem</td>
</tr>
<tr>
<td>journal</td>
</tr>
<tr>
<td>TransportRecord</td>
</tr>
<tr>
<td>client tasks</td>
</tr>
<tr>
<td>unsupportedBank (Bank);</td>
</tr>
<tr>
<td>invalidSignature (Signature);</td>
</tr>
<tr>
<td>invalidKey (Key);</td>
</tr>
<tr>
<td>reportProblem (ProblemDetails);</td>
</tr>
<tr>
<td>variables</td>
</tr>
<tr>
<td>clientParcel</td>
</tr>
<tr>
<td>clientOrigBank</td>
</tr>
<tr>
<td>clientOrigSignature</td>
</tr>
<tr>
<td>clientOrigKey</td>
</tr>
<tr>
<td>clientDestBank</td>
</tr>
<tr>
<td>clientDestSignature</td>
</tr>
<tr>
<td>clientDestKey</td>
</tr>
<tr>
<td>problemDetails</td>
</tr>
</tbody>
</table>

Figure 6.7: vanService (part 1 of 2)
protocol
try
  nested concurrent
  client: executeTransport
    (clientParcel,
     clientOrigBank,
     clientOrigSignature,
     clientOrigKey,
     clientDestBank,
     clientDestSignature,
     clientDestKey);
  // ‘Implementation’ of the provider task
  provider process
    try
      ordered sequential
        new getService (clientOrigBank, provider, theParcel);
        // Transfer the parcel from origin to destination bank
        new putService (clientDestBank, provider, theParcel);
    end sequential;
    when ProcessError :
      // Throw appropriate exception to the client
      end try;
  end process;
end concurrent;
when InvalidOriginBank :
  to client : unsupportedBank (clientOrigBank); reject;
when InvalidDestinationBank :
  to client : unsupportedBank (clientDestBank); reject;
when InvalidOriginSignature :
  to client : invalidSignature (clientOrigSignature); reject;
when InvalidDestinationSignature :
  to client : invalidSignature (clientDestSignature); reject;
when InvalidOriginKey :
  to client : invalidKey (clientOriginKey); reject;
when InvalidDestinationKey :
  to client : invalidKey (clientDestKey); reject;
when DeliveryProblem :
  provider : reportProblem (problemDetails);
end try;
journal
  client, provider, select, try, sequential, concurrent, multiple;

Figure 6.8: vanService (part 2 of 2)
in the contract. The contract for this service is very similar to the valueTransferService contract of section 6.2.1, especially with regard to the exceptions that can be thrown and the handling thereof. The vanService contract is shown in figures 6.7 and 6.8. Notes to these figures:

- The provider is of type ArmouredCar, while the client must be an instance of Transporter. The service object is the parcel to be transported. There is no default initialization here, which is different from what we have seen in the valueTransferService contract (figure 6.2). This is because the parcel is always known when the current service is instantiated.

- Like before, there is only one provider task in this contract, i.e., executeTransport (). As shown in figure 6.7, the client has to supply two client signatures as well as two keys, one to ‘get’ the goods from the origin bank, and the other to ‘put’ them at the destination bank. Obviously, the origin and destination banks themselves must also be passed.

- The protocol is extremely simple again: there is one task invocation message enclosed within an monitored interaction pattern containing handlers for the various exceptions. If the executeTransport () task ends normally, the parcel will have been moved to the destination bank. The exceptions and their handlers are virtually identical to those in figure 6.2.

- The task invocation message is followed by a process block describing its implementation. Because the process block is a specification of the implementation of the preceding task, the message and the subsequent process block must be put in a nested concurrent interaction pattern.

- Like before, most exceptions cause a client exception handler to be invoked. The client can either try to establish a new contract and provide the proper data, or it can in turn throw an exception to its own client. In case of a DeliveryProblem though, the provider has to report the details of the problem to the client by sending a reportProblem () message. This is why the last exception handler is a formal exception handler; the others were all informal handlers.

- The contract does not state what should happen if the putService were to fail after the getService has succeeded. Obviously, the driver would have to return the valuables to either the origin bank or the client.

6.2.6 getService, putService

The getService, shown in figure 6.9, is a very simple service provided by a Bank to an ArmouredCar. The contracts require the van driver to present the bank with the keys given out during execution of singlePrepareGetService or singlePreparePutService. Upon getting the client signature and the former key given by the main contractor, the armoured
car drives to the first bank and executes the `getService`. Then it drives to the second bank and delivers the goods using the client signature and the latter key. The latter action is covered by the `putService` contract.

**description**
- **domain**: ValueTransport;
- **service**: getService;

**actors**
- **provider**: Bank;
- **client**: ArmouredCar;

**objects**
- **theParcel**: Parcel;

**provider tasks**
- **executeGet**
  - (theParcel : Parcel,
    theSignature : Signature,
    theKey : Key)
  - throws InvalidSignature, InvalidKey;

**variables**
- **clientSignature**: Signature;
- **clientKey**: Key;

**protocol**
- **try**
  - client: executeGet (theParcel, clientSignature, clientKey);
  - when ValueTransport.InvalidSignature:
    - reject;
  - when ValueTransport.InvalidKey:
    - reject;
- **end try**;

**journal**
- client, provider, select, try, sequential, concurrent, multiple;

Figure 6.9: `getService`

Notes to the figure:

- The current contract describes the interaction between a bank and a physical transporter (ArmouredCar) when the latter needs to collect a parcel from a bank.
- The transporter sends a single task invocation message to the bank (`executeGet`), providing the latter with the client signature and the get-key. If something should be wrong with that information, the task will end with the appropriate exception (e.g., InvalidSignature or InvalidKey).
- Both exception handlers end with `reject`, so the contract terminates unsuccessfully if either the signature or the get-key is invalid. As the client will receive the same exception at its own business code level, it is not necessary to include a formal exception handler in the contract. The informal handler ends with a `reject`, causing the
6.2. The service contracts

contract to be ended with a ProcessError exception. The client may try to establish a new contract, this time providing valid data.

- The putService is fully equivalent to getService described above, and will not be detailed here. The only difference is that all occurrences of ‘Get’ have to be replaced by ‘Put’. In many cases, both services will be bundled into a single contract.
Chapter 6. Case Study: valueTransferService
Chapter 7

Case Study: Cross Docking

The second case study is about a cross docking station, i.e., a short-term storage for goods that come in from a variety of suppliers and have to be rearranged before they are distributed to a variety of consumers; the distribution of a multitude of (household) products from a number of different factories to a number different of supermarket chains may serve as an example (see figure 7.1). Transportation of these goods to and from the cross docking station is performed by regular trucks. The cross docking station itself is a facility that consists of a number of loading/unloading platforms as well as a small/medium scale warehouse (i.e., a number of automatic pallet stacks); see also figure 7.2. Transport within the cross-docking facility is realized by a number of ‘intelligent’ pallet trucks.

7.1 Operational aspects

In the cross docking station that we consider here, an average of about 500 pallets will be delivered daily between 3 pm and 9 pm. The number of trucks will be approximately 25, the number of unloaded pallets per truck lies somewhere between 10 and 30. The pallets will be collected for distribution the next morning between 3 am and 6 am. The average number of trucks involved will again be approximately 25.

Drivers have to pre-register themselves as well as their trucks before they arrive. They can do so by filling some web-form, by using e-mail, or by phone. Since the latter 2 methods may require the assistance of a human operator, they are effectively discouraged by a higher cost. The earlier the registration takes place, the better: timely announcement (at least

Figure 7.1: Example Cross-Docking station
one hour in advance) may result in shorter waiting times and/or lower cost. Upon arrival, the driver has to register again in order to verify his own identity and that of his truck. The pallet identities and destinations should be passed as soon as they are known (pallet identities will be verified during unloading). The pallet identities must be passed no later than at arrival time, while their destinations may be passed at a later time. Of course there is an advantage in passing them early on, as this allows for more efficient preparation for the loading and distribution process. The arrival of empty trucks, the list of pallets to be loaded, their destinations as well as the order in which they must be delivered should also be announced as early as possible. Truck and driver identities will be announced and verified too. Trucks will be unloaded in the traditional way, using electric pallet trucks operated by the truck driver. The unloading therefore takes place at traditional platforms; if the process is well-organized, it would be possible to complete this task in less than half an hour (given a maximum of 30 pallets). The loading process is comparable.

The average storage time of each pallet is about 12 hours. Since the destination of a large proportion of the pallets is not known until some hours before they must be loaded into a truck, the storage facility must support random access. Automatic rearrangement of pallets inside the warehouse would be a desirable property, since it could significantly optimize the process of preparing platform stacks for pick-up.

7.2 Design criteria

The following criteria\(^1\) should be taken as a starting point in the design of the cross docking station:

Minimization of truck handling times: Idle times before platforms become available to waiting trucks must be minimized; time to handle any necessary formalities must

\(^{1}\)Marketing-aware people may recognize these criteria as critical competitive success factors.
be minimized and overlap any waiting times, if possible; truck loading/unloading times must be minimal.

**Reliable operation:** The system should be able to diagnose possible problems in *real time*; propagation of problems must be avoided; incident handling and recovery must be very efficient and avoid delays as much as possible.

**On-line operation:** Parcel tracing and tracking must be well-supported; late modifications of arrival times and/or origins or destinations must be supported in a way that doesn’t disturb other clients and that minimizes the effect on the client that makes the change; the system must be able to interoperate with external (legacy and/or 3rd party) software systems.

**Changeability while in operation:** The system must be able to accommodate new types of trucks and to incorporate new technological developments. It must support such changes while in operation.

**Minimum cost of operation:** The system must make minimum use of human resources, have minimum maintenance cost, and make efficient use of both equipment and space.

Generally, reliability is enhanced by designing a system in a way that grants autonomy to the various actors. Dependencies are less, and each actor is free to do its job in a way that is locally optimal. Local disturbances (such as failing hardware) can be handled locally without affecting the system as a whole. Of course, each actor’s responsibilities need to be very clear, especially when it comes to problems the actor can no longer handle. The service contracts must thus cover regular as well as exceptional cases in as much detail as possible.

On-line operation and changeability are also supported by means of service contracts. Essentially, any party that implements a service contract can become involved in the operation specified by the contract. The system as a whole can define as many useful subsidiary contracts as necessary, while any party can register as a client of those contracts to obtain information. Changeability is also supported by means of service contracts: seamless transition between successive actor implementations is made possible by ongoing verification of the interaction patterns; even if the new actor doesn’t behave, at least it won’t disturb the rest of the actor community; if the actor is a vital part of the system, a consequence might of course be that the system as a whole comes to a halt (albeit in a well-defined state that can be recovered from).

Efficiency is the least well-defined notion here: local efficiency with regard to some actors might have adverse effects on global efficiency. Therefore, any design must try to find a compromise between local and global efficiency, possibly on the basis of simulation studies. The design presented in subsequent sections is based upon general ideas of what will work and what won’t; detailed simulations might help to further improve the design.
7.3 The service actors

The number of logistic actors of the cross docking station is quite large, the number of services they offer is even larger. The actors as well as their interactions are shown in the figures 7.3 to 7.8. The first set of 3 figures details the unloading process, while the second set of 3 figures details the loading process. Note that the left side of figure 7.4 is in fact identical to part of figure 7.3: to avoid repetition, the Clients have been omitted from the second figure. The same remark applies to figures 7.6 and 7.7. When looking at figures 7.4, 7.5, 7.7, and 7.8, one will note that the left part of the figures 7.4 and 7.7 (depicting the ClientManager) was omitted from the figures 7.5 and 7.8. Now let’s turn to the various actors that play a role in this design; the service contracts themselves will be described in the next section:

Client: The Clients of the cross docking system as a whole are the ‘suppliers’ and the ‘consumers’ of the transported goods. These clients own or rent the trucks that will deliver and pick up the pallets to and from the cross docking station. In order to do so, they create instances of the palletIntake (see figure 7.3) and the palletOutlet (see figure 7.6) service contracts, both of which are provided by a ClientManager.

ClientManager: Each and every interaction between Clients and the system as a whole takes place through the ClientManager (figures 7.3, 7.4, 7.6, and 7.7). In our design, there is exactly one instance of this actor, so the instances depicted in these figures all refer to the same actor. A Client interacts with the ClientManager through either an instance of the palletIntake contract (figure 7.3) or an instance of the palletOutlet (figure 7.6) contract.

The number of Clients that the ClientManager can serve simultaneously is not limited, even though only a fixed number of Clients can be physically handled at any time. In the case of the palletIntake and palletOutlet services, the ClientManager cannot handle more Clients than there are platforms; if all platforms are taken, the excess Clients will be asked to wait at a buffer parking. Though this parking itself is not controlled by the cross docking station, the trucks’ waiting takes place under control of either palletIntake or palletOutlet. Even though the ‘unloading’ phase is specified not to overlap the ‘loading’ phase, it might be better not to translate this fact into requirements for the ClientManager and to allow it to unload at one platform while loading at another. Even though this would not be done during normal operation, it might prove quite useful in the handling of exceptional situations (like pick-up trucks that suffer such extreme delays that the next round of deliveries has already begun).

The ClientManager can handle a number of active Clients simultaneously; each client corresponds to a separate handling thread, denoted by a stippled circle in the figures 7.3, 7.4, 7.6, and 7.7. Each active ‘unloading’ thread is a client of both

---

2From the perspective of the designer of the cross-docking station, the distinction between supermarket chains and their eventual customers is irrelevant.
7.3. The service actors

Figure 7.3: Cross Docking: unloading (part 1 of 3)

Figure 7.4: Cross Docking: unloading (part 2 of 3)
Figure 7.5: Cross Docking: unloading (part 3 of 3)

Figure 7.6: Cross Docking: loading (part 1 of 3)
Figure 7.7: Cross Docking: loading (part 2 of 3)

Figure 7.8: Cross Docking: loading (part 3 of 3)
truckUnloading (figure 7.3) and palletStacking (figure 7.4), while each ‘loading’ thread is a client of both truckLoading (figure 7.6) and palletUnstacking (figure 7.7). These dual client roles are depicted by the pairs of smaller circles inside the thread circles. Both truckUnloading and truckLoading are services provided by a FloorManager, while both palletStacking and palletUnstacking are services provided by a StackManager.

FloorManager: Each FloorManager (see figures 7.3 and 7.6) is responsible for whatever happens at a certain platform; there is a separate instance of this actor for every platform in the cross docking station. The services provided by a FloorManager are truckUnloading and truckLoading. However, since only one truck is allowed access to a given platform at a time, the FloorManager can only serve one instance of one of these contracts at the time.

To move pallets from a truck to a platform or vice versa, the FloorManager needs an ‘intelligent’ pallet truck (IPT). IPTs can be rented from a central TransportBroker; the corresponding rental services are called iptRental1 (for unloading) and iptRental2 (for loading). The FloorManager can rent an IPT for any period of time, dispatching it on different jobs as it sees fit. In figure 7.3 the IPT is used as a subcontractor for the multiplePalletUnload and singlePalletUnload contracts, while in figure 7.6 it is used for the multiplePalletLoad and singlePalletLoad contracts.

The TransportBroker effectively transfers control of the rented IPT to the FloorManager, which can then directly subcontract the multiplePalletUnload or multiplePalletLoad service from the IPT. In doing this, the services singlePalletUnload and multiplePalletUnload are indirectly subcontracted. Concentric circles are used in the figures 7.3 and 7.6 to visualize this: the outer circle represents the FloorManager’s client role in the iptRental1 and iptRental2 services, while the inner circle represents its client roles in the multiplePalletUnload (eq. multiplePalletUnload) and singlePalletUnload (eq. singlePalletLoad) services.

IPT: An IPT (intelligent pallet truck) is a semi-automatic (see below) vehicle that can be used to transfer pallets from a truck to a so-called platform stack or vice-versa. During this unloading/loading phase, only one IPT is allowed on the platform. The corresponding services are multiplePalletUnload and singlePalletUnload (see figure 7.3), and multiplePalletLoad and singlePalletLoad (see figure 7.6), respectively. The ‘multiple’ services are directly subcontracted by the FloorManager, while the ‘single’ services are indirectly subcontracted as part of the former services. After the unload service has completed (and before the load service has started), the platform will be closed for human operators, and one or two IPTs are used to move pallets back and forth between the platform stack and one of the automatic pallet stacks (APS); the latter moves are controlled by the moveToStack contract (see figures 7.5 and 7.8), whose client is the TransportManager. Whenever a pallet is loaded on an IPT, the pallet’s identification is checked. If possible, this should be done automatically, by reading a bar code sticker or by reading a transponder chip attached to the pallet. In case of reading errors, the driver might be instructed to verify the
7.3. The service actors

pallet ID by hand.

IPTs can operate in 3 modes: fully automatic (route and speed are controlled by the system), semi-automatic (route is controlled by the system, but speed is controlled by the driver), and manual (both route and speed are controlled by the driver). When executing the moveToStack or moveToPlatform services, the mode is fully automatic, and while executing the multiple/singlePalletUnload or multiple/singlePalletLoad services, the mode depends on the IPT’s location: at the ‘front’ of a platform (i.e., in the area that is closest to the truck), it is at the disposal of the truck driver who can use it to unload or load her truck, hence the mode is manual. Unloaded pallets must be put at the ‘back’ of the platform (the area that is opposite to the truck) in a number at locations referred to as the platform stack: these locations as well as the route to be taken to reach them are controlled by the FloorManager, hence during the actual move the mode is semi-automatic. Mode changes can be effected by the system or by the driver. The IPT has a small textual display that is used by the IPT to show messages and/or instructions to the driver: in addition, the IPT has a number of buttons the driver can use to pass status information to the system (apart from the regular direction and speed controls, of course).

As said before, IPTs are self-employed: when available for work, they register with the TransportBroker. They do so by providing the iptBrokerage service to the latter. The TransportBroker can then rent them out to the FloorManager (to execute the multiple/singlePalletUnload or multiple/singlePalletLoad services), or to the TransportManager (to execute the moveToStack or moveToPlatform services). Each IPT keeps track of its own status, and may quit if something is wrong (e.g., when a low battery or some malfunction is detected). Of course, the IPT is required to notify the party it is working for as well as the TransportBroker. The latter party will send a replacement IPT and make sure the old one is taken to a maintenance zone.

StackManager: A StackManager (figures 7.4, 7.5, 7.7, and 7.8) is responsible for storage of the pallets during the period between their delivery and pick-up. It distributes the pallets over the available automatic pallet stacks (APSs), using knowledge about pallet destinations and pick-up times, and (if known) the loading order for the truck that is going to pick up the pallets\(^3\). The ordering information may be used to determine optimal positions for the pallets to be stored and/or to determine optimal routing for the IPTs that will be used to transport the pallets from the platforms to the stacks and vice versa. There is only one instance of the StackManager for the entire cross docking station, so the actor instances depicted in the various figures all denote the same instance. The services provided by the StackManager are palletStacking (figure 7.4) and palletUnstacking (figure 7.7). Both these services are provided directly to the ClientManager.

The reason for dividing control between the ClientManager and the StackManager is to decouple the two main activities of the system (unloading/loading and stacking/un-

\(^3\)Pallets for which such information is not known will be stacked in the center of the stacks, so access times for those pallets will be longer.
stacking) from each other: the platform forms the ‘decoupling point’. By decoupling these activities, the design allows for independent optimization of the client handling services and the core service of storing the pallets. As the latter service is expected to be the bottleneck of the system as a whole, common logistic wisdom dictates that it should be in control of its own interactions; hence a central StackManager. Note that the StackManager is the sole party that generates transportation requests within the controlled area of cross docking station. The details of these transportation are left to the TransportManager though.

Like the ClientManager, the StackManager contains a number of active threads. In this case, there is a separate thread for each pallet stored. Each active ‘stacking’ thread is a client of the subcontracted services palletStoring (figure 7.4) and transferToStack (figure 7.5), whereas each ‘unstacking’ thread is a client of the subcontracted services palletRetrieving (figure 7.7) and transferToPlatform (figure 7.8). Like before, the dual client roles are depicted by pairs of small circles inside a stippled ‘thread’ circle. Both palletStoring and palletRetrieving are services provided by an APS, while the transferToStack and transferToPlatform services are provided by the TransportManager.

**TransportManager**: The TransportManager is a centralized actor that is responsible for all IPT traffic control within the cross docking station, with the exception of any IPT movements taking place at ‘open’ platforms (see above). It offers the transferToStack and transferToPlatform services to the StackManager (see figures 7.5 and 7.8). There is only one instance of this actor, so the actor instance depicted in figure 7.5 is the same as the actor instance depicted in figure 7.8. The main task of the TransportManager is to organize the transportation of pallets between platform stacks and automatic pallet stacks (APSs) and vice versa. The pallets are moved using IPTs that are under direct control of the TransportManager, which acts as a client of the IPT’s moveToStack and moveToPlatform services.

The main reason to have a centralized TransportManager is the need to avoid congestion in the area between the platforms and the automatic pallet stacks. If the IPT’s were to behave as fully autonomous actors, they would have to communicate with each other to make sure that traffic jams and delays are avoided. Furthermore, it would hardly be possible to supply every IPT with all the information needed to prioritize the various transportation jobs. In planning individual transports, the transport manager has to account for the (un)loading/transportation times on the platform, the (un)loading times at the APS, and the transportation times in the area between the platform and the APS (or vice versa). A central transport manager can be given all the knowledge it might need to assign priorities to individual transports. This is especially important during the unstacking process, when the pick-up truck may already be waiting at the platform.

To transport pallets between platforms and automatic pallet stacks, the TransportManager has to rent a number of IPTs from the TransportBroker; the corresponding service is iptRental2. It is up to the TransportManager to decide if a new rental
service is subcontracted for every pallet to be transported, for a group of pallets as a whole (one might think of a all pallets in a given platform stack), or for a certain period of time. Depending on physical layout of the docking station and the pallet stacks though, the number of IPTs that can operate simultaneously might be limited.

There is a separate thread within the TransportManager for every active iptRental12 contract. The TransportManager can then subcontract the moveToStack service to transfer pallets from the platform stack to an APS (see figure 7.5) or the moveToPlatform to move them from the APS to a new platform stack (figure 7.8). Both services can be obtained directly from the IPT. During execution of either contract, the IPT is operating in fully automatic mode.

**APS:** Each APS is an actor that corresponds to an automatic pallet stack that stores the pallets overnight. An APS provides two different services, palletStoring (figure 7.4) and palletRetrieving (figure 7.7). The client of those services is the StackManager. Pallets are delivered and picked up by IPTs at small platforms.

Since there is only one client for APSs, there is no need to use an APS broker here, as is done with IPTs. If the need might arise, APSs might be used the same way as the former resource, using a StorageBroker actor. Such an actor would be a client of an apsBrokerageService, and a provider of an apsRental service.

**TransportBroker:** The TransportBroker is a central party that manages a number of IPTs. As was explained above, an IPT is an independent party which is able to provide certain services. Clients are not allowed to contact IPT's directly, though. Instead, they contact the TransportBroker in order to get an IPT assigned to them. After that, they can sub-contract the required service from the IPT.

The main reason for having a separate TransportBroker is that there are several clients contending for IPT services: the FloorManagers and the TransportManager. Under normal circumstances, FloorManagers have priority in getting an IPT to work for them. If all the platforms are taken by unloaded pallets, however, priority might be put with the TransportManager.

In order to be able to rent out an IPT, the transport broker must have an active iptBrokerageService contract with that IPT. The TransportBroker must have a way to breach an active iptBrokerageService contract and have the IPT returned. It must also be able to replace an IPT that is rented out by another one, if some problems were reported with the former IPT.

Of all the different services shown in the figures 7.3 to 7.8, only two services are actually relevant to clients in the outside world: palletIntake and PalletOutlet. The other service contracts are only relevant for parties that implement the cross docking station. The latter class of contracts and actors might be stored in a different domain of the Logistic Contract Space, with more restricted access rights⁴.

---

⁴Currently, there are no access rights at all. The cross docking example clearly demonstrates the need to implement visibility and access rights to contracts and actors in the Logistic Contract Space.
7.4 The service contracts

This section will describe the various service contracts that were mentioned in the preceding section in more detail. The design presented above consists of two parts, the first part focuses on the pallet unloading stage, while the second part focuses on the loading stage. As the two stages are mostly symmetrical, the second stage can be described in less detail, merely highlighting things that are different from the first part.

The unloading stage is defined by the following ‘core’ service contracts (the figures in which they can be found are added between parentheses): palletIntake (7.3), truckUnloading (7.3), multiplePalletUnload (7.3), singlePalletUnload (7.3), palletStacking (7.4), palletStoring (7.4), transferToStack (7.5), and moveToStack (7.5). The loading stage is defined by the following core contracts: palletOutlet (7.6), truckLoading (7.6), multiplePalletLoad (7.6), singlePalletLoad (7.6), palletUnstacking (7.7), palletRetrieving (7.7), transferToPlatform (7.7), and finally moveToPlatform (7.7). Finally, there is a number of ‘supporting’ service contracts: iptBrokerage (7.3, 7.5, 7.6, and 7.7), iptRental1 (7.3), iptRental2 (7.5), iptRental3 (7.6), and iptRental4 (7.7). Most of these contracts are long-lived; other contracts are instantiated for the job at hand and will cease to exist when their job has been successfully completed.

7.4.1 palletIntake

The first contract is palletIntake (see figures 7.3 and 7.9 to 7.11). This contract describes the interaction that is required to get a truck unloaded at the cross docking station. The client of the contract is defined as a generic cross-docking client, the provider is the ClientManager that coordinates all deliveries and pick-ups. The initial stages of the contract all have to do with registration and verification of both the truck and the driver, activities that clearly are the responsibility of the client manager. After it has established the identities, the ClientManager assigns a platform and sub-contracts the truckUnloading service from the FloorManager that corresponds to that platform. Once the unloading process has completed, the ClientManager sub-contracts the palletStacking service from the StackManager. Even though no physical actions take place under this contract after the pallets have been stacked, the contract stays active until all unloaded pallets have left the cross docking station. Notes to figures 7.9 to 7.11:

- The ‘drop list’ is the part of the truck load that is to be ‘dropped’ (unloaded) at this cross docking station. Since the drop list contains all relevant information about the pallets to be unloaded (such as the physical pallet dimensions, the weight, the kind of load, the ownership, and the destination), it is made the service object of this contract. The service object may be left undefined at contract instantiation time, as

\[\text{Note that we have defined the drop list with respect to a single service contract; from the viewpoint of a truck driver (who may have to drop pallets at more than one location) the drop list would consist of a number of such single-contract drop lists merged together.}\]
7.4. The service contracts

// purpose : registration and transfer of pallets from truck to central stack

description
  domain : PalletTransport.CrossDocking;
  service : palletIntake;

actors
  provider : ClientManager;
  client : CrossDockingClient;

objects
  theDropList : DropList := null;

provider tasks
  registerDropList
    (theTruckID : TruckID,
     theDropList : DropList)
    pre not theDropList.isEmpty ()
    throws NoSpaceAvailable;

verifyTruck
  (theTruckID : TruckID,
   theDriverID : DriverID)
  throws UnknownTruckID, UnknownDriverID;

unloadTruck
  (theTruckID : TruckID,
   thePlatform : Platform)
  pre not theDropList.isUnloaded ()
  post theDropList.isUnloaded ()
  throws WrongPlatform;

acceptDischarge (theDropList : DropList);

client tasks
  usePlatform
    (thePlatform : Platform)
    pre thePlatform.isFree ()

freePlatform
  (thePlatform : Platform)
  pre thePlatform.isTaken (theTruckID)
  post thePlatform.isFree ()
  grantDischarge (theDropList : DropList);

waitAtParking (theWaitingTime : Time);

moveToAssignedPlatform (thePlatform : Platform);

handleInvalidTruckID (theTruckID : TruckID);

handleInvalidDriverID (theDriverID : DriverID);

Figure 7.9: palletIntake (1 of 3)
variables
theTruckID : TruckID;
theDriverID : DriverID;
theAssignedPlatform : Platform;
theActualPlatform : Platform;
theEstimatedTime : Time;

protocol
try
  ordered sequential
    // The provider has to register the truck and driver
    client : registerTruck (theTruckID, theDriverID);
    // The provider has to register the drop list, i.e., the part
    // of the truck load that is to be dropped here.
    client : registerDropList (theTruckID, theDropList);
    // Information that is missing from the drop list may be updated at a later time
unordered concurrent
  ordered sequential
    // The provider has to verify the identities of the truck and its driver;
    // the drop list is verified by the provider during unloading.
    client : verifyTruck (theTruckID, theDriverID);
    optional
      // The provider may ask the client to wait until a free platform is
      // available.
      provider : waitAtParking (theEstimatedTime);
    end optional;
    // The client has to move to the given platform
    provider : usePlatform (theAssignedPlatform);
    // After arrival at the platform, the client has to request unloading
nested concurrent
    client : unloadTruck (theTruckID, theActualPlatform);
    provider process
      { assert not theDropList.isUnloaded (); }
      // The provider delegates the unloading of the truck
      // to an unspecified (but free!) floor manager.
      new truckUnloading
        (any, provider, theDropList, theActualPlatform);
      { assert theDropList.isUnloaded (); }
    end process;
end concurrent;
// After the client has reloaded any set-aside pallets,
// it asks the provider to dismiss the truck.
client : dismissTruck (theTruckID);
// The client has to move away from to the given platform
provider : freePlatform (theActualPlatform);

Figure 7.10: palletIntake (2 of 3)
provider process
    { assert not theDropList.isStacked (); }  
    // The provider has to stack the pallets; it delegates the work to the
    // (unspecified) stack manager.
    new palletStacking (any. provider, theDropList, theActualPlatform);
    { assert theDropList.isStacked (); }
end process;
end sequential;
// Trick to update drop list info (see notes)
end concurrent;
// Pallets can not be loaded until their destinations and pickup times are known
{ assert theDropList.isComplete (); }
provider process
    { assert not theDropList.isUnstacked (); }
    { assert not theDropList.isLoaded (); }
    // The pallets in the now stacked drop list are to be unstacked and
    // loaded into new trucks on behalf of the distributing clients.
    { assert theDropList.isUnstacked (); }
    { assert theDropList.isLoaded (); }
end process;
// Some final interactions take place to wrap up the contract as a whole;
// this can only happen after all pallets have left the docking station.
provider : grantDischarge (theDropList);
client : acceptDischarge (theDropList);
end sequential;
when UnknownTruckID, UnknownDriverID :
    // Unknown trucks or drivers are not handled at all
    reject;
when NoSpaceAvailable :
    // The stacks do not have enough space to accommodate the drop list
    reject;
when TruckIDError :
    // Truck ID problems are handled by the client. Any space that was reserved
    // for the pallets must be freed.
    to client : handleInvalidTruckID (theTruckID); reject;
when DriverIDError :
    // Driver ID problems are handled by the client. Any space that was reserved
    // for the pallets must be freed.
    to client : handleInvalidDriverID (theDriverID); reject;
when WrongPlatform :
    // Driver moved to wrong platform and has to correct her mistake.
    // The contract will be resumed!
    to client : moveToAssignedPlatform (theAssignedPlatform); resume;
end try;

Figure 7.11: palletIntake (3 of 3)
specified by the **null** default value. If a drop list is passed upon contract instantiation though, the drop list passed to `registerDropList()` must be equal to the drop list passed at contract instantiation.

- The drop list may not be empty. Since the drop list service object may be initially unbound, this fact cannot be specified by means of an assertion at the beginning of the contract. Since we have included this requirement as a precondition on `registerDropList()`, the precondition will be enforced even if the drop list was already passed as the service object.

- The driver and her truck must be registered first, i.e., even before the drop list is passed. The truck ID and the driver ID must be known to the provider, or else the exceptions `UnknownTruckID` and/or `UnknownDriverID` will be thrown. Both these exceptions have an empty handler that ends with `reject`, so the contract will terminate by throwing a `ProcessError` and disallow any further communication. If the client made a mistake when passing the required information, it can make a new attempt to instantiate the contract and provide the proper information.

- After the truck and driver have been registered, the drop list can be registered by the provider. Drop list registration can take place any time between truck registration and truck verification, but preferable as early as possible: the truck may be served quicker if drop list data are available earlier on. Note that information about the destination and pick-up time frame of each individual pallet in the drop list may still be absent from the drop list when it is registered. When this happens, the information can be supplied at a later time (see below).

- Since the docking station’s stacking capacity is finite, the remaining stacking capacity might not be enough to store all pallets in the client’s drop list. When this should happen, `registerDropList()` will throw the exception `NoSpaceAvailable`. The exception has an empty handler again and ends with `reject`, so the contract is terminated immediately. A more sophisticated way to handle this exception would be to include a handler informing the client how long it would have to wait for the space to be available. The contract could then be resumed or terminated.

- The `ClientManager` verifies both the truck and its driver when they arrive at the cross docking station. If it turns out that something is wrong with either the truckID or the driverID, another exception will be thrown: `TruckIDException` or `DriverIDException`. These exceptions are serious enough to inform the client by calling one of its exception handling tasks (`handleInvalidTruckID()` and `handleInvalidDriverID()`). After that, the contract is terminated with a `reject`. Note that the provider can totally ignore the exception that it has triggered! It is the protocol verifier itself that calls the error handling tasks, taking the relevant information from the protocol state. Since the contract will be terminated anyway, the exception handling tasks do not bother to include a postcondition.
7.4. The service contracts

• As a platform might not be available yet, the provider may ask the driver to wait, while supplying an estimated waiting time. The actual end of the waiting period is signaled by the provider passing a platform to the client (i.e., by the usePlatform() following the optional wait). A variation of this would be to use a timed interaction pattern to specify a maximum waiting time.

```plaintext
timed (duration 15 Minutes)
  ordered sequential
    client: assignPlatform (...);
    optional
      provider: waitAtParking (...);
    end optional;
    provider: usePlatform (thePlatformNumber);
  end sequential;
when DurationError :
  provider : acceptCompensation (...);
  provider : usePlatform (thePlatformNumber);
end timed;
```

This protocol fragment specifies that the provider has to assign a platform to the client no later than 15 minutes after the latter has asked for one. Any waiting has to happen in between and is now constrained by the ‘timed’ construct. Even though the construct cannot not prevent excess delays from happening, it specifies that some form of compensation has to be offered to the client if they do occur. Since we still want the truck to be handled, the exception handler includes an additional invocation of the usePlatform() task. The reason we didn’t use a retry or resume to end the exception handler is that we do not know which task invocation has triggered the DurationError, and we certainly do not want to wait any longer! Note that the maximum waiting time could be specified as a service parameter to the contract.

• By invoking usePlatform(), the ClientManager assigns a platform to the truck. The driver has to move to the assigned platform and request unloading; the platform itself is passed in the request (this process could be automated by scanning the truck ID when it arrives at the platform). It would of course be possible to include the usePlatform() and the unloadTruck() in another timed interaction pattern:

```plaintext
timed (duration 15 Minutes)
  ordered sequential
    provider: usePlatform (thePlatformNumber);
    client: unloadTruck (theTruckID, theActualPlatform);
  end sequential;
when DurationError :
  reject;
end timed;
```

Unlike before, the contract is aborted if the driver doesn’t show up in time.

• Since the possibility of human errors can never be ruled out, the contract has to be prepared to handle such mistakes. In our case, the driver might show up at a different
platform than the one that was assigned to her. There are many ways to handle a situation like this. The easiest way would be to share the variable denoting the platform between `usePlatform()` and `unloadTruck()`. Assuming that the latter task invocation method is automatically generated by scanning the arriving truck (or its driver ID) at the actual platform, the `unloadTruck()` would just throw a `ProtocolViolation`. The scanning device should then present the driver with some visual clue that the unloading request was not granted. A functionally equivalent (but unnecessarily complex) solution would be to use a different ‘platform’ variable and adding a precondition to `unloadTruck()` that its value equals the assigned platform.

The preferred solution is different from both these alternatives: it specifies that `unloadTruck` might (but need not!) throw the `WrongPlatform` exception. If the actual platform is OK too, the `ClientManager` proceeds after adjusting its internal information; if it is not OK, the exception will be thrown. In the latter case a client side exception handler is invoked (by the protocol verifier!), asking the client to move to the proper platform; this could be done by displaying a message on an information display that is integrated with the scanning device. After that, the action that caused the exception is resumed. The whole process would be repeated until the driver manages to show up at the proper platform. This approach has two definite advantages: first, a client need only be corrected if that is the only way out; second, the system can do better than just signaling the problem: it can offer the solution/explanation as well by presenting a new platform number to the driver.

Since the contract is now quite forgiving with respect to possible platform errors, it is no longer possible to add a postcondition `thePlatform.isTaken()` to `usePlatform()`: it is not known which platform was actually taken until the client calls `unloadTruck()`. This is not a problem, though, as the `ClientManager` is aware of the change and knows which platforms are taken and which are still free.

- The actual unloading of the truck is subcontracted from a `FloorManager` in a process block that represents the implementation of the provider task `unloadTruck()`. The process block starts and ends with assertions that are equivalent to the pre- and postconditions of that task. Even though we assume that there will be one floor manager per platform, the specific provider of the sub-contract is not relevant at the level of the main contract. This fact is indicated by specifying `any` as the sub-contract’s provider. The client of the sub-contract is the current provider, i.e., the `ClientManager`. Note that the pre-condition on `unloadTruck()` is there for legal reasons: the `FloorManager` is responsible for all pallets in the drop list that it has unloaded, so before accepting this responsibility it needs to ensure that all pallets are actually there and have not been marked as unloaded yet! Of course it is still necessary to verify each individual pallet and make sure that all pallets in the drop list are actually present. This is done as part of the `truckUnloading` service.

- After all pallets in the drop list have been moved to the platform stack, the driver has to clear the platform and then ask to be dismissed (`dismissTruck()`). Before the `ClientManager` will reply with a `freePlatform()` request, it will have to check
(probably by means of a camera system) that the publicly accessible part of the platform is indeed empty. The \texttt{freePlatform} () request is not supposed to terminate before the truck has left the platform, hence the postcondition. Some sensor or camera system might again be used to verify this postcondition. The driver is now supposed to leave the cross docking station. Note however that the contract does not end here! This is because the client remains responsible for the unloaded pallets until they have been recollected from the cross docking station \footnote{We assume that this responsibility for a pallet cannot be transferred to a 3rd party until the pallet is checked out.}

- After the platform has been cleared by the truck driver, the \texttt{ClientManager} is responsible for having the unloaded pallets (the so-called \texttt{platform stack}) transferred to the central stacking area and having them stored there. Note that the \texttt{ClientManager} will initiate the stacking process right after the platform was cleared by \texttt{freePlatform} (); there is no need for the client to issue a request to this effect.

The process block that follows \texttt{freePlatform} () shows that the \texttt{ClientManager} delegates these jobs to an unspecified \texttt{StackManager} (there exists only one such actor, though): the delegated service is called \texttt{PalletStacking}. Since we want the drop list to be updated with detailed information concerning the stack location of every pallet to be stored as well as with precise stacking and unstacking time stamps, it is passed as a service object to the \texttt{PalletStacking} service. If the drop list entry for a given pallet already contains that pallet’s destination, the stack manager will use that information to determine the best position for the pallet. If such information is not yet known, the pallet will be put at a free position close to the middle of the central stacking area.

Even though the process block is merely a description of some process internal to the provider (instead of a more or less direct implementation of a specific provider task), we have included assertions to specify the main prerequisites and results of \texttt{palletStacking}. In particular, all pallets in the drop list will be marked as \texttt{stacked} after the sub-contract has ended. The choice of 'opening' assertions was somewhat arbitrary: we might have included an assertion to state that all pallets in the drop list had been unloaded. This addition might be useful because the process block is not coupled to a provider task. If there had been such a task, though, it would not need such a precondition as the protocol itself ensures that the task would not be executed before all pallets were unloaded!

- As some pallets’ destinations might not yet be known when the drop list is registered, there must be a way to update them at a later time. The client is therefore allowed to update the drop list after it has been registered. This can even be done after the pallets have already been stacked (see above); a consequence of setting or changing a destination after they have already been stacked is that unstacking of the affected pallets might be less efficient. An unordered concurrent construct is used to specify the time frame during which the updates to the drop list information can take place:
this period starts right after the drop list is registered, and ends after all pallets have been stacked.

There is a problem with updating the drop list though: being a service object, it cannot be modified by the contract parties! The following protocol fragment uses the protocol manager itself to do the updates, which is a reasonable workaround:

    indexed (thePalletID : PalletID)
    sequential multiple (exit : Updates)
    select
      ordered sequential
        client : updateDestination (thePalletID, theDestination);
        { update theDropList.setDestination (thePalletID, theDestination); }
      end sequential;
      client : quit (Updates);
    end select;
    end multiple;

Like before, the number of pallet info updates is not known beforehand, so we need to use an exit constraint in combination with a quit ()

- The rest of the protocol provides a sketchy description of what will happen to the pallets after they have been stacked: not only would it be difficult to describe the details of those processes, but also they are the territory of other contracts (e.g. palletOutlet). All we need to say here is that at a certain moment all pallets will have been assigned a destination, and that eventually they all will have been unstacked and picked up. One reason for stating these facts in the current contract is that after being picked up, the pallets are no longer the responsibility of the current ClientManager.

Since we want the contract to remain active until the provider has completed its tasks, we need some final interaction before we can wrap up: after all pallets in the drop list have found their way into some delivery truck, the provider is allowed to ask the client to be discharged (grantDischarge () ); the request is followed by a confirmation by the client (acceptDischarge () ).

- It should be realized that the provider process block does not reflect the actual processes that take place during unstacking and pick-up. These processes cannot be easily described here, since multiple (possibly unknown) clients are involved and since pallets are unloaded on an individual basis. The drop list does not provide any information, and pick-up lists are not yet available. The lack of information here can also be seen from the assertion that precedes the provider process block: the condition that all pallets have been assigned a destination (i.e., theDropList.isComplete () ) is really too harsh, since it requires the destinations of all pallets to be known before any pallet can be unstacked. This obviously does not reflect the reality of unstacking, but is good enough a condition in the context of the current contract.

- It might be a good idea to add a parameter section to this contract, specifying the
(exact or estimated) number of pallets to be dropped; this is especially useful if the
drop list is not yet known when the contract is instantiated.

- The protocol as shown in figure 7.10 does not require a client to pass an arrival time
when it registers the truck. Providing this information would help the \texttt{ClientManager}
in planning the various activities at the cross docking station, but at the same time
it would introduce the problem that such a time would be nothing but an estimate
that may need to be changed later. The protocol fragment shown below can be used
to allow clients to announce their arrival time.

\begin{verbatim}
ordered sequential
  client: registerTruck (theTruckID, theDriverID, theArrivalTime);

unordered concurrent
  client: registerDropList (theTruckID, theDropList);
  indexed (i : Integer)
    sequential multiple (exit : Adjustments)
    select
      client: adjustArrivalTime (i, theArrivalTimes[@i]);
      client: quit (Adjustments);
    end select;
  end multiple;
end concurrent;

client: verifyTruck (theTruckID, theDriverID);

end sequential;
\end{verbatim}

Looking at this fragment, we can identify two problems related to the multiple con-
struct: the strict need to include a multiple index into every nested call, and the
rather clumsy way to exit the multiple. Both these problems will be discussed further
in section 11.2.1.

### 7.4.2 truckUnloading

The second contract is \texttt{truckUnloading} (see figures 7.3 and 7.12). It describes the unloading
service that is offered by a \texttt{FloorManager} to the \texttt{ClientManager}. The service consists
of moving all pallets in the drop list (which is again a service object to the contract)
from the truck to the platform. The actual moving of pallets is subcontracted to an IPT;
the corresponding service contract is \texttt{multiplePalletUnload} (see section 7.4.3). The IPT
that is needed is not just magically present at the platform, but has to be rented by the
\texttt{FloorManager} from a central \texttt{TransportBroker}. After the contract has completed, all the
pallets will be positioned in a special area of the platform, the so-called \textit{platform stack}.
This area is not freely accessible to people: the driver is not supposed to be in this area
except when driving an IPT from the truck to a specific platform stack position or vice
versa. The \texttt{truckUnloading} contract is shown in figure 7.12. Notes to the figure:

- Both the drop list and the platform are used as service objects. The drop list contains
// purpose : transfer of pallets from truck to platform stack

description
  domain : PalletTransport.CrossDocking;
  service : truckUnloading;

actors
  provider : FloorManager;
  client   : ClientManager;

objects
  theDropList : DropList;
  thePlatform : Platform;

provider tasks
  unloadTruck
    (theDropList : DropList;
      thePlatform : Platform)
    pre  not theDropList.isUnloaded ()
       and thePlatform.getPlatformStack ().isEmpty ();
    post theDropList.isUnloaded ()
       and thePlatform.getPlatformStack ().getPallets ()
       == theDropList.getActualPallets ();

client tasks
  none;

protocol
  nested concurrent
    // The client has to request unloading
    client : unloadTruck (theDropList, thePlatform);

  provider process
    { assert not theDropList.isUnloaded (); }
    { assert thePlatform.getPlatformStack ().isEmpty (); }

    nested concurrent
      // The provider has to get an IPT from the transport broker
      new iptRentAll (any, provider, theDropList, thePlatform);
      // The provider has to unload the truck using the IPT it got; since we do
      // not know here which IPT this will be, 'any' is specified as the provider.
      new multiplePalletUnload
        (any, provider, theDropList, thePlatform.getPlatformStack ());

    end concurrent;

    { assert theDropList.isUnloaded (); }
    { assert thePlatform.getPlatformStack ().getPallets ()
       == theDropList.getActualPallets (); }

  end process;

end concurrent;

journal
...

Figure 7.12: truckUnloading
the pallets to be unloaded together with all relevant information about them. The platform specifies the platform at which the unloading will take place. As such, it acts as a handle to information regarding the platform, in particular to the platform stack. The platform stack is represented by a small 'database' that specifies all pallets that are stacked at the platform and the platform positions at which they are located. Since there are no default values, both service objects have to be passed at contract instantiation. Note that this contract does not require the drop list to be non-empty.

- Even though they are service objects, and thus accessible to the contract parties, both the drop list and the platform have to be passed as part of the unloading request (unloadTruck ()). The values that are passed must be identical to the values passed to the contract instantiation. The main reason to pass these values as part of the task invocation is that it allows us to specify pre- and postconditions for them. This compensates for the lack of pre- and postconditions on the service objects themselves (see section 11.2.2 for more information). The assertions specify that all (non-missing, see section 7.4.3) pallets in the drop list will be moved to the pallet stack, and that the pallet stack will not contain any pallets that were not in the drop list.

- The provider process gives a sketch of the implementation of the provider task unload-Truck, and thus contains assertions that correspond to the pre- and postconditions of that task. For reasons we have explained before, the task invocation and the process block are both put inside a nested concurrent construct.

- The process block shows that the FloorManager does not unload the truck itself, but that it sub-contracts the multiplePalletUnload service from an IPT. The IPT must be rented from the central TransportBroker by means of the iptRentAll service contract. Neither the TransportBroker nor the IPT need be known at the level of this contract, hence they are specified as any. Another nested concurrent is used to show that the subcontracted multiplePalletUnload falls entirely within the time frame of the iptRentAll contract. Note that the latter contract does not imply that a new IPT must be rented for every instance of the truckUnloading service! It is entirely within the specification for the FloorManager to rent an IPT for (say) a day and reuse it for every truckUnloading it has to provide that day.

- Note that the same multiplePalletUnload is part of a client process block inside iptRentAll (see section 7.4.18). The reason is that the TransportBroker might have different kinds of IPTs and does not want them to be used for a service that is not explicitly designated. Actually, we cannot be sure that the client side service in iptRentAll is indeed the same as the service we see here. To do that we would have to split the instantiation of a sub-contract into 2 distinct stages: instantiation and execution. The instantiation would give us a handle to the sub-contract instance that could then be passed to another provider. That provider could then verify that the client does indeed execute that particular instance of the contract. All this is not part of the current language. More about this in section 11.2.6.
• Intake checks of pallets are performed at one of the lower levels, e.g., in singlePalletUnload.

7.4.3 multiplePalletUnload

The next contract is multiplePalletUnload (see figures 7.3 and 7.13). This service is provided by an IPT; the client is the FloorManager that is responsible for the unloading process. The service consists of transferring a number of pallets from a truck to their assigned platform positions. Each individual transfer is controlled by a subcontract, singlePalletUnload (see section 7.4.4). Since both the client and the provider of this sub-contract are the same as the client and the provider of the current contract, this is not a case of delegation, and so there is no reason to put the sub-contract invocation inside a process block. Notes to figure 7.13:

• The multiplePalletUnload contract is responsible for transferring all pallets in the drop list from the truck to the platform. Since the status of the drop list and the platform stack is supposed to reflect the physical world, both need to be updated for every pallet that is moved; hence they are used as service objects. No default values are specified, so both service objects must be initialized at contract instantiation.

• Before the client can start the unloading process, it has to request the IPT to move to the proper platform area. If the IPT is already situated at that position, the provider does not have to pass the request to the IPT.

• As soon as the IPT has become available at the unloading area, the client can start the unloading process by issuing the unloadTruck () request. Both the drop list and the platform stack must be passed as arguments to the request. The values passed for these arguments must be equal to the values passed for the service objects during contract instantiation. The benefit of also passing these values as arguments to the task invocation message is that the pre- and postconditions specified in the task signature effectively become pre- and postconditions on the service objects! The precondition states that the drop list must not be unloaded yet and that the platform stack is empty. The postcondition states that all pallets in the drop list have been marked as unloaded or missing and that the platform stack contains exactly the unloaded pallets.

• If the task had been delegated to a third party, the task invocation would have been followed by a provider process block. Because the current provider is also the provider of the subcontracted service, the process block that is normally used to provide insight into the implementation of the task is replaced by an ordered sequential. Still, the task invocation message and the following sequential construct are put inside a nested concurrent construct. For completeness, assertions corresponding to the task’s pre- and postconditions are included at the beginning and the end of the sequential construct.
7.4. The service contracts

// purpose: semi-automatic transfer of multiple pallets from truck to platform stack

description

domain : PalletTransport.CrossDocking;

service : multiplePalletUnload;

actors

provider : IPT;

client : FloorManager;

objects

theDropList : DropList;

thePlatformStack : PlatformStack;

provider tasks

unloadTruck

(theDropList : DropList;
thePlatformStack : PlatformStack)

pre not theDropList.isUnloaded ()

and thePlatformStack.isEmpty ()

post theDropList.isUnloaded ()

and thePlatformStack.getPallets ()

== theDropList.getActualPallets ();

moveToBasePosition ();

protocol

ordered sequential

client : moveToBasePosition ();

nested concurrent

client : unloadTruck (theDropList, thePlatformStack);

ordered sequential

{ assert not theDropList.isUnloaded (); }

{ assert thePlatformStack.isEmpty (); }

try

indexed (thePalletIndex : Integer)

sequential multiple (theDropList.extent ())

new singlePalletUnload

(provider, client, theDropList,
thePlatformStack, thePalletIndex);

end multiple;

when ProcessError :

{ update theDropList.markMissingPallets (); }

end try;

{ assert theDropList.isUnloaded (); }

{ assert thePlatformStack.getPallets ()

== theDropList.getActualPallets (); }

end sequential;

end concurrent;

optional

client : moveToBasePosition ();

end optional;

end sequential;

Figure 7.13: multiplePalletUnload
• Each pallet in the drop list is unloaded by subcontracting a new instance of the 
  singlePalletUnload service inside a multiple construct. The multiple construct is 
  expected to have as many instances as there are pallets in the drop list (theDropList-
  .extent ()). but in the real world it might always happen that some pallets are 
  missing from the truck. The drop list and the platform stack are passed as service 
  objects to singlePalletUnload. A simple sequence number is used as the multiple 
  index. It might be instructive to have a look at an earlier attempt in defining the 
  current service contract; that version contained the following multiple:

  indexed (thePalletTag : PalletTag)
  sequential multiple (theDropList.extent ())
  { assert not theDropList.isUnloaded (thePalletTag); }
  { assert not thePlatformStack.contains (thePalletTag); }
  // Each individual pallet unload is covered by a single subcontract
  new singlePalletUnload
  (provider, client, theDropList, thePlatformStack, thePalletTag);
  { assert theDropList.isUnloaded (thePalletTag); }
  { assert thePlatformStack.contains (thePalletTag); }
  end multiple;

  The problem with that version is that the pallet tag has to be known when the sub-
  contract is instantiated; it is not possible to use default initializations for service 
  indexes. As a result, the pallet order has to be known in advance. This requirement 
  is of course difficult to realize, especially if during an earlier drop the driver had to 
  put some of our pallets aside and load them afterwards.

• In case there are pallets missing from the truck, one of the subcontracted single-
  PalletTransfer services is bound to fail. The best we can do is making the sub-
  contract throw a ProcessError exception if it cannot find any more pallets that 
  are present in the drop list. If singlePalletTransfer ends with this exception, we 
  therefore know that all pallets that have not been marked as unloaded are missing 
  from the truck; the update in the exception handler marks the ProcessError marks the 
  remaining pallets as missing in the drop list. We haven't included any assertions in 
  the exception handler here, since they are in fact implied by the update. They would 
  have looked as follows:

  { assert theDropList.isUnloaded (); }
  { assert thePlatformStack.getPallets () == theDropList.getActivePallets (); }

  We assume that getMissingPallets () returns the set of pallets that have been 
  marked as missing, and that getActualPallets () returns all non-missing pallets 
  from the drop list. The missing pallets have to be reported to the client, of course, 
  but to save some space this exception handling detail was omitted from figure 7.13.

• Since the truck may contain pallets that have been set aside during the unloading 
  process (see also section 7.4.4), the driver may need the IPT to reload them afterwards.
The (optional) invocation of `modeToBasePosition()` instructs the IPT to move back to the public area.

### 7.4.4 singlePalletUnload

The `singlePalletUnload` contract (see figures 7.3 and 7.14) describes the transfer of a single pallet from the truck to a platform stack position. Even though it would have been possible to integrate the contents of this contract with the higher level `multiplePalletUnload`, keeping them separate improves readability. Both the provider and the client of `singlePalletUnload` are identical to the provider and the client of the higher level contract. The provider, i.e., the IPT operates in two different modes: operator controlled in the public area of the platform, and system controlled in the restricted area of the platform (see also section 7.3). The latter area is where the platform stack positions are located. Notes to figure 7.14:

- The `singlePalletUnload` contract describes how a single pallet is unloaded by an IPT. The service objects are the same as those of the `multiplePalletUnload` contract (see section 7.4.3). Again, no default values are specified. Since the service is subcontracted from inside a multiple construct, a multiple index (`thePalletIndex`) must be passed to the current service contract instance. The presence of this service index forces us to include a `thePalletIndex` argument to all task invocation messages exchanged in this contract (and therefore to the task signatures of all those tasks). The index itself does not have any intrinsic meaning; it is just a number that serves to distinguish the different multiple branches.

- Because the pallet tag is scanned by the IPT when the pallet is unloaded from the truck, it is not possible to specify any assertions with respect to that specific pallet until after `loadPallet()` has terminated. It is therefore impossible to start the contract with assertions that act like preconditions, or to put such preconditions on `loadPallet`.

- Pallets are scanned and verified during loading by `loadPallet()`. Two problems might occur when executing this task: the scanned pallet ID might not be present in the drop list, or there might be no more pallets left to load. In the former case, `loadPallet()` will throw an `UnknownPallet` exception. The IPT will display a message in its display, asking the driver to put the pallet aside. The exception handler for `UnknownPallet` can be empty except for a retry that will send the protocol to the beginning of the try construct, ready for unloading the next pallet. After all pallets in the drop list have been moved to their respective stack positions, the truck driver may use the IPT to reload the pallets that were temporarily set aside back into her truck. In the latter case, `loadPallet()` must throw a `ProcessError` exception; this is the only exception that need not be handled in the contract and that will be propagated to the higher level contract. When this exception is thrown, we may conclude that the remaining pallets in the drop list are missing.
Chapter 7. Case Study: Cross Docking

// purpose: semi-automatic transfer of single pallet from truck to platform stack
description
  domain  : PalletTransport.CrossDocking;
  service : singlePalletUnload;
actors
  provider : IPT;
  client   : FloorManager;
objects
  theDropList : DropList;
  thePlatformStack : PlatformStack;
indexes
  thePalletIndex : Integer;
provider tasks
  loadPallet
    (thePalletIndex : Integer,
     theDropList : DropList)
    throws UnknownPallet;
  unloadPallet
    (thePalletIndex : Integer,
     thePosition : PlatformPosition);
client tasks
  doneLoadingPallet
    (thePalletIndex : Integer,
     thePalletTag : PalletTag)
    pre theDropList.contains(thePalletTag);
  doneUnloadingPallet (thePalletIndex : Integer);
variables
  thePalletTags : map [Integer] of PalletTag;
  thePlatformPositions : map [Integer] of PlatformPosition;
protocol
try
  ordered sequential
    // The pallet tag is obtained during loading
    client : loadPallet (thePalletIndex, theDropList);
    provider : doneLoadingPallet (thePalletIndex, thePalletTags[thePalletIndex]);
    { assert not theDropList.isUnloaded (thePalletTags[thePalletIndex]); }
    { assert not thePlatformStack.contains (thePalletTags[thePalletIndex]); }
    client : unloadPallet (thePalletIndex, thePlatformPositions[thePalletIndex]);
    provider : doneUnloadingPallet (thePalletIndex);
    { update theDropList.setUnloaded (thePalletTags[thePalletIndex]); }
    { update thePlatformStack.add
      (thePalletTags[thePalletIndex],
       thePlatformPositions[thePalletIndex]); }
end sequential;
when UnknownPallet :
  // Unknown pallets are set aside to be reloaded later
  retry;
end try;

Figure 7.14: singlePalletUnload
7.4. The service contracts

- If no problems turned up during pallet verification and loading, the driver has to signal the end of the loading process by pressing a button on the IPT. This action will trigger the client task `doneLoadingPallet ()`, which is merely a signal to the client that the provider is ready to accept a new command. Note that `doneLoadingPallet ()` has a precondition stating that only pallets in the drop list can get this far: the handling of unknown pallets is not controlled by the current contract. At this stage of the unloading process, we can finally update the service objects. The assertions verify that the IPT has loaded a pallet that has not yet been marked as unloaded, and that the pallet stack does not already contain a pallet with the same tag.

The client then requests the IPT to perform a controlled move to the unloading position (`unloadPallet ()`): the driver can control the speed of the IPT, but not its route. After it has reached the target position, the IPT will switch back to manual operation and notify the driver that she can unload the pallet. The IPT might still monitor its own movements though, and restrict the driver to a limited area around the intended pallet position. By no means it will allow unloading of the pallet at the wrong platform position! The driver has to signal the end of the unloading process by pushing a button again, thus triggering another ‘handshaking’ task to the client (`doneUnloadingPallet ()`). The IPT moves back to its base position as part of the next `loadPallet ()` request. Because of this, the execution of `singlePalletUnload` will leave the IPT at the last unloading position.

- After the pallet was deposited at the platform position, the service objects of the contract are updated to reflect the new situation; this update is guaranteed to fulfill the assertions at the end of the multiple. Note that although the update is part of the contract itself, it might not get executed until the next interaction takes place at the level of the ‘parent’ contract. Formally, the current contract does not terminate until the update has taken place.

7.4.5 palletStacking

The `palletStacking` service contract (figures 7.4 and 7.15) describes the second half of the intake process: the transfer of pallets from a platform to the central pallet stack (the APS), and their storage in that stack. The transfer is subcontracted from the `TransportManager` (transferToStack, see section 7.4.7), while the storing of a pallet is subcontracted from the APS (`palletStoring`, section 7.4.6). Both these subcontracted services apply to individual pallets, even though the `palletStacking` service still applies to all the pallets in the drop list that are not marked as missing, i.e., precisely the pallets that are in the platform stack. Notes to figure 7.15:

- This contract follows the pattern of the others: both the drop list and the platform are used as service objects, even though they are also passed as arguments to `storePallets ()`. Since there are no default values, the service objects have to be initialized at contract instantiation. The values passed in the `storePallets ()` mes-
Chapter 7. Case Study: Cross Docking

// purpose: transferring/storing of pallets in the platform stack

description
  domain: PalletTransport.CrossDocking;
  service: palletStacking;

actors
  provider: StackManager;
  client: ClientManager;

objects
  theDropList: DropList;
  thePlatform: Platform;

provider tasks
  storePallets
    (theDropList: DropList,
     thePlatform: Platform)
    pre
      not theDropList.isStacked ()
    and
      thePlatform.getPlatformStack ().getPallets ()
      == theDropList.getActualPallets ()
    post
      theDropList.isStacked ()
    and
      thePlatform.getPlatformStack ().isEmpty ()

protocol
  nested concurrent
    client: storePallets (theDropList, thePlatform);

  provider process
    ordered sequential
      { assert not theDropList.isStacked (); } 
      { assert thePlatform.getPlatformStack ().getPallets ()
        == theDropList.getActualPallets (); }

    indexed (thePalletTag: PalletTag)
    concurrent multiple (thePlatform.getPlatformStack ().getSize ()
      ordered sequential
        { assert thePlatform.getPlatformStack ().contains (thePalletTag); }
      ordered concurrent
        new transferToStack
          (any, provider, thePlatform.getPlatformStack (),
           thePalletTag);
        new palletStoring (any, provider, theDropList, thePalletTag);
      end concurrent;
      end sequential;
      end multiple;
    end process;
  end concurrent;

  journal
...

Figure 7.15: palletStacking
sage must be identical to the values passed for the service objects during contract instantiation. The precondition to storePallets () states that the pallets in the drop list must not yet be stacked, and that the platform stack must contain all pallets from the drop list that have not been found missing. The postcondition states that all pallets that were in the platform stack have now been stacked in the APS.

- The provider process is a partial specification of the implementation of the storePallets () task. It starts and ends with assertions that are equivalent to the pre- and postconditions of storePallets (). The pallets are handled on an individual basis in the body of a concurrent multiple. Concurrency is needed because more than one pallet is being stored at the same time, and because more than one IPT might be involved in the transport of pallets between the platform and APS: the platform allows for 2 IPTs, and even more might be active in the area between the platform and the automatic pallet stacks. The pallet tags are used to index the multiple.

- Both the storage and the transfer of a pallet are subcontracted by the provider: to the TransportManager (transferToStack) and to the APS (palletStoring), respectively. Both transferToStack and palletStoring are per-pallet services. The former service does has the pallet stack as its service object, while the latter service has the original drop list as its service object. Since the pallet tag is used as a multiple index, it must be passed as a service index to both subcontracted services.

- Even though there will be a specific timing relation between transferToStack and palletStoring, the contract can hardly be specific here, since it does not know when the pallet will arrive at the APS and if there might be a time overlap between the two subcontracted services (most likely there is). The ordered concurrent construct is flexible enough to handle all possible orderings.

- The transport of pallets from platform to APS is subcontracted from the TransportManager. The transferToStack contract does not state a provider, but of course we know that there is only one TransportManager in the system.

- The contract does not specify which APS will be the provider of the palletStoring service for a specific pallet: the algorithm to allocate stacking space to pallets is private to the StackManager. Most likely the stack manager will use information about the destination of each pallet (if already known), maybe even information that can be derived from the destination, e.g., the platform at which the pallet has to be loaded into a delivery truck.

7.4.6 palletStoring

The PalletStoring service (figures 7.4 and 7.16) is concerned with the temporary storage of pallets in an automatic pallet stack (APS). The provider is an APS, while the client of the service must be the StackManager. The contract is instantiated for every individual pallet to be stored.
// purpose: automatic stacking of a pallet by an APS

description
  domain: PalletTransport.CrossDocking;
  service: palletStoring;

actors
  provider: APS;
  client: StackManager;

objects
  theDropList: DropList;

indexes
  thePalletTag: PalletTag;

provider tasks
  storePallet
    (thePalletTag: PalletTag)
    pre not theDropList.isStacked (thePalletTag)
      and not provider.contains (thePalletTag)
    post theDropList.isStacked (thePalletTag)
      and provider.contains (thePalletTag);

protocol
  ordered sequential
  client: storePallet (thePalletTag);
    { update theDropList.setStacked (thePalletTag); } end sequential;

journal
  ...

Figure 7.16: palletStoring
7.4. The service contracts

Notes to figure 7.16:

- Even though the service applies to a single pallet, the original drop list is used as a service object. The reason is that we want to mark the pallet as stacked in the drop list. Since the contract is used within a multiple construct, the pallet tag must be passed as a service index to the contract.

- There has been a lot of discussion as to whether it would be nice to be able to use assertions to state that the APS must not be full before a pallet is stored, that it will not be empty after the pallet has been stored, and that it cannot be full anymore after the pallet has been retrieved. According to Evers [Eve04], such conditions belong to the process logic of the provider, i.e., the APT. The problem with not stating them, though, is that it remains unclear how e.g., a full APS situation should be handled, if it occurs. Should it be impossible to instantiate the PalletStoring service? Should the service be instantiated but not allow the storePallet () task to be executed? Should it allow that task to be executed, but delay its execution until the pallet can indeed be stored? These are all questions to be answered before the actor processes can be implemented, so if contracts are to serve as a specification for the implementor of the logistic actors, we think they should be complete.

There is still a problem with assertions though: they are often not adequate in the context of concurrent execution. Consider the case of the APS, which is designed to provide many instances of the service simultaneously. The assertions might look as follows:

```plaintext
ordered sequential
{ assert not provider.isFull (); }
client : storePallet (thePalletTag):
{ assert not provider.isEmpty (); }
end sequential;
```

The problem is that although the APS could have a free position when the assertion is checked, that position might be taken by the time the pallet must be stored. This would even be the case if the conditions were stated as pre- and postconditions on storePallet (given the usual semantics of pre- and postconditions). In the latter case it would be possible though to change the semantics of pre- and postconditions in such a way that they would act as waiting conditions (as is done in Eiffel [Mey97]) that also claim the resources after they became available. A similar coupling between waiting and claiming should be designed for assertions (see also section 11.2.7).

7.4.7 transferToStack

The transferToStack service (see figures 7.5, 7.17 and 7.18) realizes the transportation of pallets between platform stacks and central pallet stacks. The provider is a TransportManager. Many transports might be active concurrently, and several of them might need
the same resources. These shared resources not only include the IPTs used to transport the pallets, but also the floor space needed to make the necessary maneuvers and the access to loading/unloading areas of the centralized pallet stacks. Since the total throughput of the system is very important, a centralized **TransportManager** that is aware of all transportation requests is preferred over a society of autonomous IPTs: all transports are controlled directly by this **TransportManager**, which subcontracts IPTs to have the transports done.

```plaintext
// purpose: transfer of a pallet from platform to APS
description
  domain   : PalletTransport.CrossDocking;
  service  : transferToStack;
actors
  provider : TransportManager;
  client   : StackManager;
objects
  thePlatformStack : PlatformStack;
indexes
  thePalletTag : PalletTag;
provider tasks
  transferToStack
    (thePalletTag : PalletTag,
     thePlatform  : Platform,
     thePosition  : Position,
     theAPS       : APS)
    pre  thePlatform.getPlatformStack () == thePlatformStack
         and thePlatformStack.getPosition (thePalletTag) == thePosition
    post  thePlatform.getPlatformStack () == thePlatformStack
         and not thePlatformStack.contains (thePalletTag)
    throws WrongPallet;
client tasks
  handleWrongPallet
    (thePalletTag : PalletTag,
     thePlatform  : Platform,
     thePosition  : Position);
variables
  thePlatform     : Platform;
  thePosition     : map [PalletTag] of Position;
  theAPS          : map [PalletTag] of APS;
```

Figure 7.17: transferToStack (part 1 of 2)

Notes to figure 7.17 and 7.18:

- The purpose of **transferToStack** is to have a single pallet transferred between the platform stack and the automatic pallet stack (APS). Because the platform stack is affected by the transfer, it is passed as the service object. Since the service is subcontracted from within a multiple construct, the **thePalletTag** must be passed
protocol
  try
    nested concurrent
      client: transferToStack
        (thePalletTag,
         thePlatform,
         thePosition[@thePalletTag],
         theAPS[@thePalletTag]);
    provider process
      ordered sequential
        { assert thePlatformStack.getPosition (thePalletTag)
        == thePosition[@thePalletTag]; }
        new iptRental2 (any, provider, thePalletTag);
        new moveToStack (any, provider, thePlatformStack, thePalletTag);
        { assert not thePlatformStack.contains (thePalletTag); }
      end sequential;
    end process;
  end concurrent;
when WrongPallet :
  to client: handleWrongPallet
    (thePalletTag,
     thePlatform,
     thePosition[@thePalletTag]);
  reject;
end try;

Figure 7.18: transferToStack (part 2 of 2)
as a service index.

- The client has to invoke `transferToStack`(), providing the pallet tag and both an origin and a destination of the transfer. The precondition of `transferToStack`() specifies that the pallet must indeed be stored at the specified position of the given platform. The postcondition states that the pallet is no longer in the platform stack. Perhaps surprisingly, `transferToStack`() may throw the `WrongPallet` exception. This exception is necessary because there might be a discrepancy between the virtual world as given by `thePlatform` and the physical world. This discrepancy could be caused by malicious intent or by a software problem. To catch such problems, the IPT scans each pallet when it is loaded and throws a `WrongPallet` exception if the scanned pallet tag differs from the expected tag. The corresponding exception handler invokes an exception handling task on the client and ends by rejecting the entire contract.

- Contrary to `thePosition` and `theAPS`, the variable `thePlatform` was not declared as a `map`. This because we wanted to make clear that all instances of this contract that are created under the same multiple must apply to the same platform.

- A provider process provides a sketch of the implementation of `transferToStack`(). As usual, the block starts and ends with assertions that correspond to the task's pre- and postconditions. The body of the process block follows the pattern we have already seen in `truckUnloading` (see section 7.4.2): an IPT is obtained from the `TransportBroker` under the `iptRental12` service and subsequently used as a provider of `moveToStack`. Note that both sub-contracts have `thePalletTag` as a service index; this is a consequence of `thePalletTag` being an index of the `transferToStack` service itself.

### 7.4.8 moveToStack

The `moveToStack` (see figures 7.5 and 7.19) service is used to physically transfer pallets from a platform to an APS position. Compared to the `singlePalletUnload` service of section 7.4.4 or the `singlePalletLoad` service of section 7.4.16, the IPT operates in only one mode (automatic), which simplifies this contract to little more than a single task invocation message. Notes to figure 7.19:

- The pallet stack is passed to the contract as a service object since is must be updated after the pallet was removed. Since the service is indirectly subcontracted from inside a multiple construct, `thePalletTag` is declared as a service index.

- The main difference between the 'general' move and the platform moves specified by `multiplePalletUnload/singlePalletUnload` and `multiplePalletLoad/single-

---

[^7]: A named exception is used since one of the possible causes lies outside the system itself. If a software error would be the only possible cause, it would have been more reasonable to throw a `ProcessError` exception instead.
7.4. The service contracts

// purpose : transfer of single pallet from platform stack position to an APS

description
  domain : PalletTransport.CrossDocking;
  service : moveToStack;

actors
  provider : IPT;
  client : TransportManager;

objects
  thePlatformStack : PlatformStack;

indexes
  thePalletTag : PalletTag;

provider tasks
  transferToStack
    (thePalletTag : PalletTag,
     thePlatform : Platform,
     thePosition : Position,
     theAPS : APS)
  pre thePlatform.getPlatformStack () == thePlatformStack
  and thePlatformStack.getPosition (thePalletTag) == thePosition
  throws WrongPallet;

client tasks
  handleWrongPallet
    (thePalletTag : PalletTag,
     thePlatform : Platform,
     thePosition : Position);

variables
  thePlatform : Platform;
  thePosition : map [PalletTag] of Position;
  theAPS : map [PalletTag] of APS;

protocol
  try
    ordered sequential
      client : transferToStack
        (thePalletTag,
         thePlatform,
         thePosition[@PalletTag],
         theAPS[@PalletTag]);

        { update thePlatformStack.setRemoved (thePalletTag); }

    end sequential;
  when WrongPallet :
    to client : handleWrongPallet (thePalletTag, thePlatform, thePosition[@PalletTag]);
    reject;
  end try;

journal
...

Figure 7.19: moveToStack
PalletLoad is that there is no driver involved in general moves. Unloading and loading of pallets is fully automatic, as is the routing between the loading and the unloading positions. The IPT is controlled by the client (i.e., the TransportManager), who can use global knowledge about the stacking process to avoid traffic congestion. Low-level control is performed by the IPT itself.

- The task to be executed by the IPT is invoked by transferToStack(). The pallet tag is passed as the index argument, the platform and the platform position specify the origin of the move, and the APS specifies the destination of the move. As in the preceding contract, the platform is the same for services that are subcontracted under the same multiple, so thePlatform does not need to be indexed by thePalletTag. The precondition states that the origin position does indeed contain the given pallet. Note that it is not possible to add a postcondition stating that the origin position is empty afterwards! This is because the provider can not update the service object itself, and has to rely on the service contract to do this via an update. This update follows the task invocation message in an ordered sequential. (One might be tempted to use a nested concurrent to specify that the update must take place during the execution of the preceding task; this is not allowed though: an update must always be preceded and/or followed by the exchange of a message under control of a sequential construct).

- Since the IPT validates the pallet tag while loading the pallet, we have to be prepared to handle a WrongPallet exception. Of course this is highly unlikely, and can only be the result of illegal pallet moves at a platform that was supposed to be closed for humans. Software malfunction might also come to one's mind as a possible cause for this exception. The exception has to be handled by the client, i.e., the StackManager. Both the pallet tag and the platform position are passed to the client exception handler.

7.4.9 palletOutlet

The next series of contracts starts with palletOutlet (see figures 7.6 and 7.20 to 7.22). The palletOutlet contract is the opposite of the palletIntake service described in section 7.4.1; it describes the interaction that is required to load a specific set of pallets into a truck. The client of the contract is defined as a generic cross-docking client, the provider is the ClientManager that coordinates all deliveries and pick-ups. The initial stages of the contract all have to do with registration and verification of both the truck and the driver, activities that clearly are the responsibility of the client manager. Notes to figure 7.22:

- The pick-up list is the opposite of the drop list in the earlier contracts. It contains the tags of all pallets to be loaded for distribution. There is no additional information in the pick-up list, since all relevant information was already present in the original drop list or added to that list while the pallets were stored in the APS. Note that the
7.4. The service contracts

// purpose : transfer of pallets from central stack to truck
description
  domain : PalletTransport.CrossDocking;
domain : palletOutlet;

actors
  provider : ClientManager;
  client : CrossDockingClient;

objects
  thePickupList : PickupList := null;

provider tasks
  registerPickupList
    (theTruckID : TruckID,
     thePickupList : PickupList)
  pre not thePickupList.isEmpty ()
  throws UnknownPallets, UnknownPalletData;
  registerTruck
    (theTruckID : TruckID,
     theDriverID : DriverID)
  throws UnknownTruckID, UnknownDriverID;
  verifyTruck
    (theTruckID : TruckID,
     theDriverID : DriverID)
  throws TruckIDError, DriverIDError;
  loadTruck
    (theTruckID : TruckID,
     thePlatform : Platform)
  pre not thePickupList.isLoaded ()
  post thePickupList.isLoaded ()
  throws WrongPlatform;
  dismissTruck (theTruckID : TruckID);
  acceptDischarge (thePickupList : PickupList);

client tasks
  usePlatform
    (thePlatform : Platform)
  pre thePlatform.isFree ()
  freePlatform
    (thePlatform : Platform)
  pre thePlatform.isTaken (theTruckID)
  post thePlatform.isFree ()
  grantDischarge (thePickupList : PickupList);
  waitAtParking (theWaitingTime : Time);
  moveToAssignedPlatform (thePlatform : Platform);
  handleInvalidTruckID (theTruckID : TruckID);
  handleInvalidDriverID (theDriverID : DriverID);

Figure 7.20: palletOutlet (1 of 3)
variables
    theTruckID : TruckID;
    theDriverID : DriverID;
    theAssignedPlatform : Platform;
    theActualPlatform : Platform;
    theEstimatedTime : Time;

protocol
    try
        ordered sequential
        // The provider has to register the truck and driver
        client : registerTruck (theTruckID, theDriverID);
        // The provider has to register the pick-up list, i.e., the set
        // of pallets that is to be collected.
        client : registerPickupList (theTruckID, thePickupList);
        unordered concurrent
            provider process
                { assert not thePickupList.isUnstacked (); }
                // The provider has to un-stack the pallets; it delegates the work to the
                // (unspecified) stack manager.
                new palletUnstacking (any, provider, thePickupList, theAssignedPlatform);
                { assert thePickupList.isUnstacked (); }
            end process;
            ordered sequential
                // The provider has to verify the identities of the truck and its driver;
                // the pick-up list is verified by the provider during loading.
                client : verifyTruck (theTruckID, theDriverID);
                optional
                    // The provider may ask the client to wait until the platform is
                    // available.
                    provider : waitAtParking (theEstimatedTime);
                end optional;
        end sequential;
    end concurrent;
    // The client has to move to the given platform
    provider : usePlatform (theAssignedPlatform);

Figure 7.21: palletOutlet (2 of 3)
7.4. The service contracts

// After arrival at the platform, the client has to request loading

nested concurrent
client : loadTruck (theTruckID, theActualPlatform):

provider process
{ assert not thePickupList.isLoaded (): }

// The provider delegates the loading of the truck
// to an unspecified (but free!) floor manager.
new truckLoading (any, provider, thePickupList, theActualPlatform):
{ assert thePickupList.isLoaded (): }

end process;
end concurrent;

// After the client has cleared any set-aside pallets from the platform,
// it asks the provider to dismiss the truck.
client : dismissTruck (theTruckID);

// The client has to move away from the given platform
provider : freePlatform (theActualPlatform);

// Some final interactions take place to wrap up the contract as a whole:
// this can only happen after all pallets have left the docking station.
provider : grantDischarge (thePickupList);
client : acceptDischarge (thePickupList);

end sequential;

when UnknownTruckID, UnknownDriverID :
// Unknown trucks or drivers are not handled at all
reject;

when UnknownPallets, UnknownPalletData :
// One or more pallets cannot be found in the system, or are lacking information.
reject;

when TruckIDError :
// Truck ID problems are handled by the client. Any space that was reserved
// for the pallets must be freed.
to client : handleInvalidTruckID (theTruckID); reject;

when DriverIDError :
// Driver ID problems are handled by the client. Any space that was reserved
// for the pallets must be freed.
to client : handleInvalidDriverID (theDriverID); reject;

when WrongPlatform :
// Driver moved to wrong platform and has to correct her mistake.
// The contract will be resumed!
to client : moveToAssignedPlatform (theAssignedPlatform); resume;
end try;

journal
...

Figure 7.22: palletOutlet (3 of 3)
pick-up list can still contain pallets that are not present in the APS or pallets whose destination is not yet known. An exception will be thrown in both these cases, and the contract will be aborted. Note that the pick-up list may contain pallets for more than one destination!

- Since the pallets will be unstacked shortly before the truck will arrive at the cross docking station, it is not possible to change platform after the system has started moving pallets to the platform. If a truck appears at the wrong platform, the exception WrongPlatform will be thrown. The formal handler specifies that he will have to move to the proper platform and try again.

- It is assumed that the truck will always have enough space to take all pallets in the pickup list. Obviously, the contract could be extended to cover the situation that the truck can't pick up all pallets whose pick up it it had announced. The handler for this exceptional situation would probably specify that pallets have to be stored again. Alternatively, the client could be forced to send an additional truck to pick up the remaining pallets as soon as possible.

- The two process blocks correspond to one of the process blocks that were already specified in palletIntake. The subcontracted services in that contract are exactly the same service contract instances as the service contract instances in the current contract.

- It might be a good idea to merge the palletIntake and palletOutlet service contracts into a single multi-party contract. Doing so would reduce the overlap between both contracts and clarify some of the interactions between pallet intake and outlet. Integration of the two service contracts would also be necessary to make the drop list reflect the pick-up of pallets. As the contracts are now, the assertions in palletIntake regarding unstacking and pick-up can never succeed as the original drop cannot be updated from the current contract. More research is necessary to determine a proper way to handle mutual interdependencies like these.

### 7.4.10 palletUnstacking

The palletUnstacking service contract (figures 7.7 and 7.23) describes the first half of the outlet process: the retrieving of pallets from the central pallet stack (the APS) and their transportation to a platform. The retrieving of a pallet is subcontracted from the APS (palletRetrieving, section 7.4.11), while its transport is subcontracted from the TransportManager (transferToPlatform, see section 7.4.7). Both these subcontracted services apply to individual pallets, while the palletUnstacking service applies to all the pallets in the pick-up list. Notes to figure 7.23:

- The differences with the pallet stacking contract (section 7.4.5) are very minor. The multiple construct is constrained by the size if the pick-up list instead of the platform
// purpose : retrieving/transfering of pallets in the platform stack

description
  domain : PalletTransport.CrossDocking;
  service : palletUnstacking;
actors
  provider : StackManager;
  client : ClientManager;
objects
  thePickupList : PickupList;
  thePlatform : Platform;
provider tasks
  retrievePallets
    (thePickupList : PickupList, thePlatform : Platform)
    pre
      thePickupList.isStacked ()
    and thePlatform.getPlatformStack ().isEmpty ()
    post
      thePickupList.isUnstacked ()
    and thePlatform.getPlatformStack ().getPallets () == thePickupList.getPallets ()

protocol
  nested concurrent
    client : retrievePallets (thePickupList, thePlatform);

provider process
  ordered sequential
  { assert thePickupList.isStacked (); }
  { assert thePlatform.getPlatformStack ().isEmpty (); }
  indexed (thePalletTag : PalletTag)
  concurrent multiple (thePickupList.getSize ())
    ordered sequential
    ordered concurrent
    new palletRetrieving
      (any, provider, thePickupList, thePalletTag);
    new transferToPlatform
      (any, provider, thePlatform.getPlatformStack (), thePalletTag);
    end concurrent;
  
    { assert thePlatform.getPlatformStack ().contains (thePalletTag); }
  end sequential;
end multiple;

  { assert theCollectlist.isUnstacked (); }
  { assert thePlatform.getPlatformStack ().getPallets () == thePickupList.getPallets (); }
end sequential;
end process;
end concurrent;

journal ...

Figure 7.23: palletUnstacking
stack (which is the result of this contract). Also, the assertions referring to the platform stack and the APS have been exchanged, while the assertion in the body of the construct was moved to the end. The subcontracted services have been replaced by services that arrange the opposite pallet movement.

7.4.11 palletRetrieving

The PalletRetrieving service (figures 7.7 and 7.24) is concerned with the retrieval of stored pallets from an automatic pallet stack (APS). The provider is an APS, while the client of the service must be a StackManager. The contract is instantiated for every individual pallet to be retrieved.

```java
// purpose : automatic unstacking of a pallet by an APS
description
  domain   : PalletTransport.CrossDocking;
service   : palletRetrieving;
actors
  provider : APS;
  client   : StackManager;
objects
  thePickupList : PickupList;
indexes
  thePalletTag : PalletTag;
provider tasks
  retrievePallet
    (thePalletTag : PalletTag)
    pre thePickupList.isStacked (thePalletTag)
    and provider.contains (thePalletTag)
    post thePickupList.isUnstacked (thePalletTag)
    and not provider.contains (thePalletTag);
protocol
  ordered sequential
    client   : retrievePallet (thePalletTag);
    { update thePickupList.setUnstacked (thePalletTag); } end sequential;
journal
...
```

Figure 7.24: palletRetrieving

Notes to figure 7.24:

- The drop list has been replaced by the pick-up list, the 'store' task of the complementary contract has been replaced by a 'retrieve' task, and the assertions have been adapted to the change. For the rest, palletRetrieving is identical to the
palletRetrieving contract described in section 7.4.6.

7.4.12 transferToPlatform

The transferToPlatform service (see figures 7.8) and 7.25 realizes the transportation of pallets between central pallet stacks and platform stacks. The service is the opposite of transferToStack (see section 7.4.7). Notes to figure 7.25:

- The main difference with the complementary contract of section 7.4.8 is the absence of the WrongPlatform exception and the corresponding handler. This exception was not necessary here, since the pallets stored in the APS can not be replaced without the system knowing it. So if a wrong pallet is detected, it must be a software error. Such errors should not be associated with a named exception; a ProcessError is a better solution.

7.4.13 moveToPlatform

The moveToPlatform (see figures 7.8 and 7.26) service is used to transfer pallets from an APS position to a platform. Compared to the singlePalletUnload service of section 7.4.4 or the singlePalletLoad service of section 7.4.16, the IPT operates in only one mode (automatic), which simplifies this contract to nothing more than a single task invocation message. Notes to figure 7.26:

- The changes in this contract are similar to those in the previous contract.

7.4.14 truckLoading

The next contract is truckLoading (see figures 7.6 and 7.27). The service is the opposite of the truckUnloading service described in section 7.4.2. The new service consists of moving all pallets in the pick-up list (which is a service object to the current contract) from the platform to the truck. Notes to figure 7.27:

- The truckLoading service contract is very similar to the truckUnloading contract of section 7.4.2: unloadTruck () has been replaced by loadTruck (). Compared to the former task, the pre- and postconditions of the latter have been exchanged; the same is true for the assertions at the beginning and end of the process block.

- The pick-up list always contains exactly the pallets that have to be loaded; a pick-up list containing pallets that are not present in the APS is simply rejected. The pallets in the pick-up list can be retrieved by getPallets (). There is no equivalent of the getActualPallets () and getMissingPallets () methods. The decision not to load the available pallets and inform the client of any missing ones was taken because
// purpose: transfer of a pallet from APS to platform

description
  domain: PalletTransport.CrossDocking;
  service: transferToPlatform;

actors
  provider: TransportManager;
  client: StackManager;

objects
  thePlatformStack: PlatformStack;

indexes
  thePalletTag: PalletTag;

provider tasks
  transferToPlatform
    (thePalletTag: PalletTag,
     theAPS: APS,
     thePlatform: Platform,
     thePosition: Position)
    pre
    thePlatform.getPlatformStack() == thePlatformStack
    and not thePlatformStack.contains(thePalletTag)
    post
    thePlatform.getPlatformStack() == thePlatformStack
    and thePlatformStack.getPosition(thePalletTag) == thePosition;

variables
  thePlatform: Platform;
  thePosition: map of [PalletTag] Position;
  theAPS: map of [PalletTag] APS;

protocol
  nested concurrent
    client: transferToPlatform
      (thePalletTag,
       theAPS[@thePalletTag])
      thePlatform,
      thePosition[@thePalletTag]);

provider process
  ordered sequential
    { assert not thePlatformStack.contains(thePalletTag); }
    new iptRental3(any, provider, thePalletTag);
    new moveToPlatform(any, provider, thePalletTag);
    { assert thePlatformStack.getPosition(thePalletTag)
      == thePosition[@thePalletTag]; }

  end sequential;
  end process;
  end concurrent;

Figure 7.25: transferToPlatform
// purpose: transfer of single pallet from an APS to a platform stack position

description
  domain: PalletTransport.CrossDocking;
  service: moveToPlatform;

actors
  provider: IPT;
  client: TransportManager;

objects
  thePlatformStack: PlatformStack;

indexes
  thePalletTag: PalletTag;

provider tasks
  transferToPlatform
    (thePalletTag: PalletTag,
     theAPS: APS,
     thePlatform: Platform,
     thePosition: Position)
    pre
      thePlatform.getPlatformStack() == thePlatformStack
      and not thePlatformStack.contains(thePalletTag);

client tasks
  none;

variables
  thePlatform: Platform;
  thePosition: map [PalletTag] of Position;
  theAPS: map [PalletTag] of APS;

protocol
  ordered sequential
    client: transferToStack
      (thePalletTag,
       theAPS[@thePalletTag],
       thePlatform,
       thePosition[@thePalletTag]);

    { update thePlatformStack.add
      (thePalletTag,
       thePosition[@thePalletTag]); }

  end sequential;

journal
  ...

Figure 7.26: moveToPlatform
// purpose : transfer of pallets from platform stack to truck

description
  domain : PalletTransport.CrossDocking;
  service : truckLoading;

actors
  provider : FloorManager;
  client : ClientManager;

objects
  thePickupList : PickupList;
  thePlatform : Platform;

provider tasks
  loadTruck
    (thePickupList : PickupList;
     thePlatform : Platform)
    pre not thePickupList.isLoaded ()
    and thePlatform.getPlatformStack ().getPallets ()
      == thePickupList.getPallets ();
    post thePickupList.isLoaded ()
    and thePlatform.getPlatformStack ().isEmpty ();

client tasks
  none;

protocol
  nested concurrent
    // The client has to request loading
    client : loadTruck (thePickupList, thePlatform);

  provider process
    { assert not thePickupList.isLoaded (); }
    { assert thePlatform.getPlatformStack ().getPallets ()
      == thePickupList.getPallets (); }

  nested concurrent
    // The provider has to get an IPT from the transport broker
    new iptRental4 (any, provider, thePickupList, thePlatform);

    // The provider has to load the truck using the IPT it got; since we do
    // not know here which IPT this will be, 'any' is specified as the provider.
    new multiplePalletLoad
      (any, provider, thePickupList, thePlatform.getPlatformStack ());

  end concurrent;

    { assert thePickupList.isLoaded (); }
    { assert thePlatform.getPlatformStack ().isEmpty (); }

  end process;

  end concurrent;

journal
...
in this case we know exactly which pallets are present in the stacks; this is different
from the truckUnloading contract, because in that case we didn’t know which pallets
were missing until the entire truck had been unloaded.

- Like before, an IPT is obtained from the TransportBroker to do the actual moving.
The IPT has to be rented as before: the iptRental13 service described in section 7.4.20
is almost identical to the iptRental11 service of section 7.4.18.

7.4.15 multiplePalletLoad

The next contract is multiplePalletLoad (see figures 7.3 and 7.28). This service is pro-
vided by an IPT; the client must be a FloorManager. The service consists of physically
transferring a number of pallets from their respective platform positions to the truck. Each
individual transfer is controlled by a subcontract, singlePalletLoad (see section 7.4.16).
Notes to figure 7.28:

- Since we can be certain that the stack doesn’t contain any unknown pallets, the
multiple can use the pallet tag as its index. This is different from the contract of
section 7.4.3 which had to ‘invent’ a multiple index.

- Since the IPT needs the truck driver to handle the loading/unloading process, it has
to move to the platform’s base area before the loading can start. If some pallets have
to be taken out of the truck before the pallets are loaded, the driver is allowed to use
the IPT to set them aside. Of course, these pallets have to be moved back to the truck
after the new pallets have been loaded. Since the IPT is already at the base position,
there is no need to request the IPT to move back.

7.4.16 singlePalletLoad

The singlePalletLoad contract (figures 7.3 and 7.29) is the opposite of singlePlatform-
Unload (see section 7.4.4). Notes to the figure:

- The service consists of transferring pallets from a platform into a truck. Since all
pallets in the platform stack have originated from the APS, there is no need to deal with
unknown pallets, like in the singlePalletUnload contract. Still, loadPallet ()
might see itself confronted with a pallet that it didn’t expect given the location: if
this happens, it can only signal a serious software problem. Since the handling of
software errors is outside the scope of service providers and/or clients, it is not very
useful to have loadPallet () generate a regular named exception; it is better to
generate a ProcessError instead.
// purpose : semi-automatic transfer of multiple pallets from platform stack to truck
description
  domain     : PalletTransport.CrossDocking;
  service    : multiplePalletLoad;
actors
  provider   : IPT;
  client     : FloorManager;
objects
  thePickupList  : PickupList;
  thePlatformStack : PlatformStack;
provider tasks
  loadTruck
    (thePickupList  : PickupList;
     thePlatformStack : PlatformStack)
  pre not thePickupList.isLoaded ()
    and thePlatformStack.getPallets ()
    == thePickupList.getPallets ();
  post thePickupList.isLoaded ()
    and thePlatformStack.isEmpty ()
  moveToBasePosition ();
protocol
  ordered sequential
  client : moveToBasePosition ();
  nested concurrent
  client : loadTruck (thePickupList, thePlatformStack);
  ordered sequential
    { assert not thePickupList.isLoaded (); }
    { assert thePlatformStack.getPallets ()
      == thePickupList.getPallets (); }
  indexed (thePalletTag : PalletTag)
  sequential multiple (thePickupList.extent ())
    new singlePalletLoad
      (provider, client, thePickupList, thePlatformStack, thePalletTag);
  end multiple;
    { assert thePickupList.isLoaded (); }
    { assert thePlatformStack.isEmpty (); }
  end sequential;
end concurrent;
end sequential;

Figure 7.28: multiplePalletLoad
// purpose: semi-automatic transfer of single pallet from platform stack to truck

description
  domain: PalletTransport.CrossDocking;
  service: singlePalletLoad;

actors
  provider: IPT;
  client: FloorManager;

objects
  thePickupList: PickupList;
  thePlatformStack: PlatformStack;

indexes
  thePalletTag: PalletTag;

provider tasks
  loadPallet
    (thePalletTag: PalletTag,
     thePosition: PlatformPosition);
  unloadPallet (thePalletTag: PalletTag);

client tasks
  doneLoadingPallet (thePalletTag: PalletTag);
  doneUnloadingPallet (thePalletTag: PalletTag);

variables
  thePlatformPositions: map [PalletTag] of PlatformPosition;

protocol
  ordered sequential
    { assert not thePickupList.isLoaded (thePalletTag); }
    { assert thePlatformStack.contains (thePalletTag); }
    client: loadPallet (thePalletTag, thePlatformPositions[@thePalletTag]);
    provider: doneLoadingPallet (thePalletTag);
    client: unloadPallet (thePalletTag);
    provider: doneUnloadingPallet (thePalletTag);
    { update thePickupList.setLoaded (thePalletTag); }
    { update thePlatformStack.setRemoved (thePalletTag); }
end sequential;

journal
...

Figure 7.29: singlePalletLoad
7.4.17 iptBrokerage

The IPTs that are used to move pallets from one position to another are independent actors. They can not be contracted directly, though, and are assigned to a particular job by the TransportBroker. Clients may obtain an IPT by instantiating a rental contract with the transportBroker. In order to be able to rent out a certain IPT to perform a given job for a client, the transport broker has to be a client of that IPT first. The corresponding service is iptBrokerage (see figures 7.3, 7.5, 7.6, 7.7, and 7.30). Notes to figure 7.30:

- IPTs are independent providers that establish a relation with a central Transport-Broker. The latter actor is the client of the iptBrokerage service, while the IPT is the provider. There is no service object involved in this service.

- The protocol is very simple: the TransportBroker has to send an acceptOwner () request to the IPT, passing an actor that will become the temporary owner of that IPT. Since the temporary owner is passed as a parameter, a variable must be declared to hold the actor value. Since pallet trucks are only rented out to either the TransportManager or the FloorManager, the variable is declared as a LinCBasic-Actor, which is the common ancestor of both these actors.

- The assignment is followed by a process block showing that one of four possible sub-contracts is instantiated with the IPT as its provider. The temporary owner is passed as the client of the subcontracted service.

There is a problem with the subcontracted iptRental contracts: it is impossible to specify any service objects, parameters, or indexes, since they are not known in the current context: instead they are determined by theOwner when it instantiates the contracts. This hints of a more general problem with sub-contracts that are instantiated within process blocks: the process block is only a partial specification and lacks the context of the actual sub-contract instantiation; in particular, the service objects and parameters might not be available. The problem might be solved by separating sub-contract instantiation from sub-contract reference (see section 11.2.6).

- After the sub-contract has terminated, the TransportBroker can discharge the IPT from working for its temporary owner.

7.4.18 iptRental1

The iptRental1 contract (see figures 7.3 and 7.31) describes the rental of a single IPT from the TransportBroker by a FloorManager client. The IPT will be used to unload a single truck at a specified platform. Notes to figure 7.31:

- The contract treats the drop list and the platform as service objects; this is because they have to be used in the service that is delegated by the client to the rented IPT.
7.4. The service contracts

// purpose: rental of a IPT's to (basic) logistic actors in the current domain

description
  domain: PalletTransport.CrossDocking;
  service: iptBrokerage;
actors
  provider: IPT;
  client: TransportBroker;
objects
  none;
provider tasks
  acceptOwner
    (theOwner : LineBasicActor)
    pre not hasOwner ()
    post getOwner ().equals (theOwner);
renounceOwner
    (theOwner : LineBasicActor)
    pre getOwner ().equals (theOwner)
    post not hasOwner ();
client tasks
  none;
variables
  theOwner : LineBasicActor;
protocol
  ordered sequential
    // The provider has to accept commands from the specified owner.
    client : acceptOwner (theOwner);
    provider process
      select
        new iptRental1 (provider, theOwner, see text);
        new iptRental2 (provider, theOwner, see text);
        new iptRental3 (provider, theOwner, see text);
        new iptRental4 (provider, theOwner, see text);
      end select;
    end process;
    // The provider has to reject commands from the specified owner.
    client : renounceOwner (theOwner);
  end sequential;
journal
...

Figure 7.30: iptBrokerage
// purpose: rental of an IPT to a floor manager to perform multiplePalletUnload

description
  domain : PalletTransport.CrossDocking;
  service : iptRental1;

actors
  provider : TransportBroker;
  client : FloorManager;

objects
  theDropList : DropList;
  thePlatform : Platform;

provider tasks
  assignIPT
    (thePlatform : Platform)
    pre client.getPlatform() == thePlatform
       and not thePlatform.hasIPT();
  freedIPT
    (theIPT : IPT)
    pre thePlatform.hasIPT()
    post not thePlatform.hasIPT();

client tasks
  useIPT
    (theIPT : IPT)
    pre not thePlatform.hasIPT()
    post thePlatform.hasIPT();

variables
  theIPT : IPT;

protocol
  ordered sequential
    // The provider has to assign an IPT to the given platform
    client : assignIPT (thePlatform);
    // The client has to use the assigned IPT
    provider : useIPT (theIPT);
    client process
      // This is the same instance as that of figure 7.12!
      new multiplePalletUnload
        (theIPT, client, theDropList, thePlatform.getPlatformStack());
    end process;
    // The client has to return the IPT after usage
    client : freedIPT (theIPT);
  end sequential;

journal
...

Figure 7.31: iptRental1
7.4. The service contracts

- When asking the TransportBroker to assign an IPT, the FloorManager has to pass the platform it is responsible for; the value that is passed must be equal to the service object. The precondition to assignIPT () states that the client must indeed be associated with the platform that is passed as the task argument. The precondition also specifies that the platform does not already have an IPT assigned to it: this is necessary to ensure that there is only one IPT active at the platform during unloading. Note that this also prevents multiple instantiations of the iptRental1 contract to circumvent the restriction of a single IPT per platform. At the same time it prevents the simultaneous instantiation of the iptRental2 service at the same platform.

- The TransportBroker replies with a message that includes the IPT that is assigned to the client. To avoid routing conflicts with other IPTs in the system, this IPT is directed to the platform by the TransportManager (the service contract that takes care of this is not included in this case study). The client can simply wait until the IPT has arrived. It would be most convenient if useIPT () would not return until the IPT is present at the platform (a postcondition could be included to enforce this).

- Both useIPT () and freeIPT () could be extended with pre- and postconditions ensuring that only a free IPT is assigned to the platform. Such conditions were omitted because the IPT itself is supposed to make sure that it is assigned to one job at the time. It wouldn’t harm to include this requirement in the form of conditions, of course.

- The actual job to be executed by the rented IPT is specified as a subcontracted service inside a client process block. It is important to note that the sub-contract instance is the very same instance as the multiplePalletUnload instance in figure 7.12! The drop list and the platform stack are passed as service object to the subcontracted service. Actually, the need to specify a drop list here is the sole reason why the iptRental1 contract has this service object.

- The contract could be modified to describe a rental period that covers the unloading of more than one truck; however, this would prevent the same IPT to be used by a subsequent iptRental2 service whose task is to transfer the unloaded pallets from the platform to an APS.

- There should be a possibility for an IPT to quit the job in case of mechanical problems. This should be messaged to the transport manager, which can then allocate another IPT. The current contract has no such provision.

7.4.19 iptRental2

The iptRental2 contract (see figures 7.5 and 7.32) describes the rental of a single IPT from the TransportBroker by a TransportManager. The IPT will be used to move all pallets from a specified platform to the central automatic pallet stack (APS). During automatic
operation, more than one IPT is allowed at a platform, so there may exist more than one instance of this contract at the same time.

```plaintext
// purpose : rental of an IPT to the transport manager to perform moveToStack
description
  domain : PalletTransport.CrossDocking;
  service : iptRental2;
actors
  provider : TransportBroker;
  client : TransportManager;
objects
  none;
indexes
  thePalletTag : PalletTag;
provider tasks
  assignIPT (thePlatform : Platform);
  freeIPT (theIPT : IPT) pre thePlatform.hasIPT ();
client tasks
  useIPT (theIPT : IPT) post thePlatform.hasIPT ();
variables
  thePlatform : Platform;
  theIPT : IPT;
protocol
  ordered sequential
    // The provider has to assign an IPT
    client : assignIPT ();
    // The client has to use the assigned IPT
    provider : useIPT (theIPT);
    client process
      // This is the same instance as that of figure 7.18!
      new moveToStack (theIPT, client, thePalletTag);
    end process;
    // The client has to return the IPT after usage
    client : freeIPT (theIPT);
  end sequential;
journal
```

Figure 7.32: iptRental2

Notes to figure 7.32:

- The `iptRental2` contract is quite similar to `iptRental1` (section 7.4.18). The new contract does not have any service objects though. Another difference is the service index, which is necessary because `iptRental2` is subcontracted from within a multiple construct.
7.4. **The service contracts**

- When asking the `TransportBroker` to assign an IPT, the `TransportManager` has to pass a platform; unlike what we saw in the `iptRental1` contract, the platform is not a service object; neither is it passed to the subcontracted service. The only reason the client has to pass a platform is that it helps the provider to select an IPT: most likely the same IPT that was used to unload the truck.

- The rest of the contract is identical to `iptRental1`.

### 7.4.20 `iptRental3`

The `iptRental3` contract (see figures 7.8 and 7.33) describes the rental of a single IPT from the `TransportBroker` by a `TransportManager`. The IPT will be used to move all pallets from the central automatic pallet stack (APS) to a specified platform. During automatic operation, more than one IPT is allowed at a platform, so there may exist more than one instance of this contract at the same time.

Notes to figure 7.33:

- The only difference with figure 7.32 is that the subcontracted service `moveToStack` has been replaced by `moveToPlatform`.

### 7.4.21 `iptRental4`

The `iptRental4` contract (see figures 7.6 and 7.34) describes the rental of a single IPT from the `TransportBroker` by a `FloorManager` client. The IPT will be used to load a single truck at a specified platform.

Notes to figure 7.34:

- The only differences with figure 7.31 are that the subcontracted service `multiplePalletUnload` has been replaced by `multiplePalletLoad` and that the `theDropList` has been replaced by the `thePickupList`.
// purpose : rental of an IPT to the transport manager to perform moveToPlatform
description
  domain : PalletTransport.CrossDocking;
service : iptRental3;
actors
  provider : TransportBroker;
  client : TransportManager;
objects
  none;
indexes
  thePalletTag : PalletTag;
provider tasks
  assignIPT (thePlatform : Platform);
  freeIPT (theIPT : IPT) pre thePlatform.hasIPT ()
client tasks
  useIPT (theIPT : IPT) post thePlatform.hasIPT ()
variables
  thePlatform : Platform;
  theIPT : IPT;
protocol
  ordered sequential
    // The provider has to assign an IPT
    client : assignIPT ();
    // The client has to use the assigned IPT
    provider : useIPT (theIPT);
    client process
      // This is the same instance as that of figure 7.25!
      new moveToPlatform (theIPT, client, thePalletTag);
  end process;
    // The client has to return the IPT after usage
    client : freeIPT (theIPT);
  end sequential;
journal
...

Figure 7.33: iptRental3
7.4. The service contracts

// purpose : rental of an IPT to a floor manager to perform multiplePalletLoad
description
    domain : PalletTransport.CrossDocking;
    service : iptRental4;
actors
    provider : TransportBroker;
    client : FloorManager;
objects
    thePickupList : PickupList;
    thePlatform : Platform;

provider tasks
    assignIPT
        (thePlatform : Platform)
        pre client.getPlatform () == thePlatform
           and not thePlatform.hasIPT ();
        freeIPT
            (theIPT : IPT)
            pre thePlatform.hasIPT ()
            post not thePlatform.hasIPT ();

client tasks
    useIPT
        (theIPT : IPT)
        pre not thePlatform.hasIPT ()
        post thePlatform.hasIPT ();
variables
    theIPT : IPT;

protocol
    ordered sequential
        // The provider has to assign an IPT to the given platform
        client : assignIPT (thePlatform);
        // The client has to use the assigned IPT
        provider : useIPT (theIPT);
        client process
            // This is the same instance as that of figure 7.27!
            new multiplePalletLoad
                (theIPT, client, thePickupList, thePlatform.getPlatformStack ());
        end process;
        // The client has to return the IPT after usage
        client : freeIPT (theIPT);
    end sequential;

journal
    ...

Figure 7.34: iptRental4
Part IV

Implementing the LinC platform
In the fourth part of this thesis we will provide a sketch of the prototype implementation of the LINC platform. As a part of this sketch, we will provide insight into how logistic contracts can be turned into something that can be used during execution of the logistic processes, both as a means to validate their interaction and as a means to gain insight into the progress made by those processes. Whereas part III has shown that it is possible to specify realistic logistic systems by defining a set of logistic interaction contracts, our aim here is to show that these contracts can indeed be used to validate the interactions in real-time, and that doing so provides a wealth of information that can be used to monitor progress and to provide a reliable (maybe even legal) basis to settle possible disagreements between the parties involved in the interaction. We will also show how logistic contracts can be used to locate and contact possible parties to cooperate with, thus enabling the self-organizing society of actors we have outlined in part I of this thesis.

The LINC platform is based upon the assumption that logistic actors are being modelled by software objects representing those actors; the interaction between logistic actors is then modelled by communication between those objects. Obviously, the way logistic systems are modelled is not so much different from distributed systems in general, so when implementing a software platform for logistic contracting one would certainly want to build upon available knowledge in that field. In chapter 8 we will therefore show how distributed systems have evolved over recent years. In particular, this chapter presents an overview of possible interaction mechanisms and touches upon the pro's and con's of each one. In conclusion, one of the interaction mechanisms will be selected as the basis for the LINC platform.

Finally, chapter 9 will show how the LINC platform can build naturally upon the selected Java RMI/Jini platform\(^8\). This chapter will explain how that platform was used to allow a logistic actor to discover any actors offering services it is interested in, how contracts can be compiled into a structure that can be used to verify interaction in real time, and how the LINC compiler might be supplemented by tools to support the lookup/discovery process\(^9\). The discussion about a possible implementation will be sketchy, though, and highlight only those aspects that we think are necessary to demonstrate that the LINC language can in fact be implemented on the basis of the chosen interaction mechanism. Also, one should bear in mind that the presented prototype implementation is only one way to build a LINC implementation. The presented architecture is expected to stay intact if a different framework is chosen, though. Understanding the LINC implementation will clear up many of the design decisions that were presented as-is in many of the rationale sections of part II. We did of course try to limit dependencies on a specific implementation when making those choices; some dependencies (such as the requirement that service objects are Remote and task arguments are Serializable (if not Remote)) were inevitable, of course.

---

\(^8\)Readers who want to gain a quick understanding of the basics of Java RMI and Jini are referred to appendix A; this appendix provides enough information to allow reading the rest of part IV without needing to resort to the references.

\(^9\)Prototype implementations of the compiler and the journal tree browser have been built and used for demonstration purposes; the other tools described in this chapter have not yet been implemented.
Chapter 8

Distributed computing

It has been assumed from the very beginning of the development of AgileFrames that logistic actors would be implemented by software objects (in the O-O sense) [ELL00, Eve00]. Admittedly, such an assumption is likely to cause a certain bias with regard to the possible mechanisms for implementing communication between those objects. To avoid this pitfall, the interaction mechanisms that have been considered for LINC include message based interaction, method based interaction, as well as communication through a shared data space. Indeed, all these mechanisms have been used in the implementation of both TrAcEs and FoRcEs, the two other components of AgileFrames [Lin03].

In order to understand the choices involved, it is necessary to present the general concepts related to distributed programming. Following a short introduction into the evolution of distributed systems (section 8.1), subsequent sections will introduce the interaction mechanisms upon which distributed systems may be based (section 8.2). When a mechanism is implemented on top of some ‘lower level’ mechanism\(^1\), the order in which they will be presented is such that the lower level is explained before the higher level mechanism that uses it. This presentation will be followed by a short discussion about the transparency of systems for remote method invocation (section 8.3) and the role of interface definition languages (section 8.4). These discussions are relevant since the LINC language incorporates and builds upon an interface definition language of its own. Moreover, since the actors of logistic contracts will ultimately be implemented by ‘regular’ code, designers of such code should be aware of the specific problems of writing distributed code and of the semantics of remote method invocations in particular: implementing the actors is just as difficult as implementing any object intended to be used in a distributed environment. As the LINC platform also defines how actors may locate and contact each other, a section on object discovery is included (section 8.6). Finally, a particular mechanism will be selected to form the basis for the pilot implementation of LINC (section 8.7). As a final note, we should mention that the information in this chapter is only intended as a survey of available techniques; it is neither complete nor does it cover the entire depth of every subject involved. It does, we hope, provide enough information to appreciate the many decisions that were made during the design of the services language and the implementation of a pilot compiler.

\(^1\)The quotes around ‘lower level’ are meant to indicate that this does not mean that those mechanisms are in some way worse and are to be avoided; they can in fact be quite useful in their own right.

219
8.1 Evolution of distributed systems

The historical development of distributed systems clearly reflects the capabilities of the hardware on which they were run. Whereas mainframe systems gave rise to large monolithic systems, the advent of workstations and personal computers spurred client-server systems: systems in which a central mainframe was used as a (database) server by a large number of local clients running query and update applications. This allowed both types of system to exploit their capacities to the maximum: very large, sometimes parallel applications on mainframes, and elaborate graphical user interfaces where most of the processing is related to user input and output on the workstations. Even though such client/server systems constituted a large step forward in terms of hardware and maintenance cost, little was gained in terms of flexibility as the clients and servers were still strongly coupled to each other and since they were not based upon open standards. This prevented 3rd parties to develop compatible clients and servers.

Client/server architectures subsequently evolved into the more general concept of a service. Services generalize the idea of programs, clients, servers, users, files, or indeed anything that may supply, record, and/or process data. Such services may depend on other services, so the provider of one service may act as the client of another service. Every complete system therefore consists of a set of services that can be used together in order to perform a particular task. Clients may locate the services they need by means of some kind of naming and directory service. An example of this kind of system is the Jini architecture (see also section A.2); the Jini design document [AOS+99] introduces the system as follows:

"The purpose of the Jini architecture is to federate groups of devices and software components into a single, dynamic, distributed system. The resulting federation provides the simplicity of access, ease of administration, and support for sharing that are provided by a large monolithic system, while retaining the flexibility, uniform response, and control provided by a personal computer or workstation. ... The dynamic nature of a Jini system enables services to be added or withdrawn from a federation at any time according to the demand, need, or changing requirements of the workgroup using the system."

Even though the Jini architecture is often explained in terms of service providers and clients [Edw00, OW00], there is no real distinction between those roles. However, both Jini providers and Jini clients implement some well-defined functionality that can be relied upon in the interaction between the actors, and usually there is exactly one instance of each actor to supply that functionality.

With the explosive growth of Internet, new kinds of distributed systems have come into vogue. Many such systems drop the distinction between clients and servers entirely. Instead, the contributing systems are referred to as peers. An interesting aspect of such systems is that they are based upon spontaneous (self-organizing) networking, and that the 'providers' are in fact distributed over a group of peers. Many instant messaging (chat) services and
(music) file sharing services are based upon this so-called P2P (peer-to-peer) paradigm. Examples are the ICQ messaging service [ICQ] and the Gnutella file sharing service [GNU]. Whereas ICQ is still a hybrid service that relies on both client/server and P2P techniques, Gnutella is a pure P2P application.

8.2 Interaction mechanisms in distributed systems

Basically, all distributed systems are based upon the exchange of messages between processes running on different systems. Message based mechanisms are extremely general; they can be found in various forms in different operating systems and environments and form the basis of most networking environments (e.g., TCP/IP). Enterprise systems often use higher level messaging software (e.g. JMS [MHD00]) that is able to keep messages to processes that are not reachable or running; such messages will be delivered after the destination has become reachable again. Message based interaction will be described briefly in section 8.2.1. An indirect way to exchange messages is by means of a shared data space where parties can wait for/read messages that have been left there by other parties. This mechanism, which is essentially a distributed version of a whiteboard system, has been formalized in the form of the Linda [Gel85] language, and implemented by e.g. JavaSpaces [FHA99]. The latter system will be introduced in section 8.2.2. While being general, messaging systems are also somewhat low-level; as such they are often used to implement higher level mechanisms such as Remote Procedure Calls (RPC’s) [Sri95a]. The RPC mechanism allows one process to call procedures and/or functions that are executed as part of another process; it hides most of the complexity of inter-process communication by making remote procedure calls look similar to local calls. RPC based interaction will be introduced in section 8.2.3. Remote procedure calls have been generalized to remote method invocations in distributed object based platforms (e.g., Java RMI [RMI, Gro01]). Distributed objects will be the topic of section 8.2.4. More recently, yet another form of interprocess communication has gained interest: P2P (peer-to-peer). JXTA is a good example of the P2P paradigm [JXTa, OTG02, Wil02]. Section 8.2.5 will describe the latter paradigm in some more detail.

8.2.1 Message based interaction

Message based systems allow one process to send a message to another process that is receiving messages; these processes do not need to run on the same machine. Messaging systems exist in many variations, mostly with regard to reliability and synchronization. The abstraction level may vary too: modern enterprise systems use a higher level messaging system than network applications that have been around for decades. A widely accepted abstraction of message based inter-process communication is CSP (Communicating Sequential Processes [Hoa78]).

In virtually every kind of messaging system connections may be either point-to-point or
broadcast type. These are the only types supported by most network level (e.g. IP) messaging systems. Some higher level systems like the Java Messaging System (JMS) [MHD00] also allow a variation known as the publisher/subscriber model; in such a system messages are sent to a specific set of parties that have expressed interest in those messages. Usually, communication may be synchronous (i.e., the processes synchronize at the message exchange, and both the send and the receive operation are blocking), or asynchronous (the send operation is non-blocking, and the receive operation may either be blocking or non-blocking). In the case of asynchronous communication, most high level systems (e.g., JMS) do not require the receiving process to be reachable (or even running, for that matter) when the message is sent; in that case the send operation will enqueue the message. As soon as the receiving party becomes reachable again, it will receive any pending messages destined for it. Most high-level messaging systems are language-independent; this is usually achieved through the use of XML [HM01, McL01] to describe the form and contents of the data to be exchanged. One such system is SOAP (Simple Object Access Protocol), which offers both message and method based interaction [Eng02, McL01].

Another form of message based interaction is found in distributed events, a form of events that can be transmitted between different processes. As with regular events, the publisher/subscriber model is used to determine which parties the event will be sent to. Interested parties are notified by calling the appropriate callback procedure or method. This method must be a remote procedure or method, of course (see also sections 8.2.3 and 8.2.4). Jini distributed events are an example of this model [OW00]; like with all other Jini mechanisms, subscribers have to obtain a lease in order to get notified of such events (see also section A.2). An advantage of the distributed event mechanism is that it makes it relatively easy to connect heterogeneous systems [CDK01].

Communication between processes is inherently unreliable: network packets may be corrupted or dropped altogether, the order in which packets are received may be different from the order in which they were sent, or a given packet might be received more than once. Most network level protocols handle most of these problems as they arise, so applications only need to deal with problems that couldn’t be solved at the network level. Usually, the main problem that remains to be handled at the application level is the total loss of packets and hence of the message these packets were part of. Unfortunately, packet loss cannot be distinguished from long delays, so a protocol that uses retransmits must be clever enough to handle possible duplicates. Similar considerations apply to distributed events, whose delivery semantics may vary: some applications may be able to operate with unreliable transmission causing some events to be dropped, whereas others may require that all events are indeed delivered. A more strict requirement might be that events are delivered in order, exactly once, and possibly even in real-time. In contrast, some systems allow events to be stored and forwarded at a later time.
8.2. Interaction mechanisms in distributed systems

8.2.2 Interaction through a shared data space

It is possible to define an indirect form of messaging through a shared data space in which one party may leave messages that can be subsequently retrieved from the space by other parties. The mechanism is somewhat similar to high level messaging, but allows for stronger synchronization than the latter mechanism. The mechanism was first defined in the form of the Linda language [Gel85], which is considered an early player in the field of coordination languages [GC92, Arb98, PA98, Row01, RCD01]. In this section we will introduce JavaSpaces [FHA99, OW00], which is a good example of a practical implementation of a shared data-space.

JavaSpaces can be viewed as a persistent store for arbitrary Java objects; it is implemented as a regular Jini (see section A.2) service. The store supports four primitive operations: objects can be written into a JavaSpace, read from a JavaSpace, taken (i.e., read and removed) from a JavaSpace, and listeners can register to be informed when objects are written into a space. The ‘read’ and ‘take’ operations can be either blocking or unblocking. The listener API can also be used to achieve synchronization between parts of a distributed program.

8.2.3 Remote procedure calls

While being general and flexible, message based interaction mechanisms are not suitable for every kind of application. In particular, messages represent data rather than control, and to use them to transfer control from one process to another in a distributed algorithm requires control information to be encoded in the form of messages. The easiest way to accomplish this is to lift the interprocess communication to the level of the programming language and to automate the transition to the message level that is still present ‘under the hood’. This mechanism is commonly referred to as remote procedure calls. One of the earliest references to RPC’s is [Nie81].

Remote procedure calls have been part of Unix based operating systems since the mid eighties: they were already part of e.g., SunOS in 1986, although standardization did not happen until much later [Sri95a]. The mechanism, called RPC, relied on particular coding conventions and was supported by a compiler that generated most of the actual code to make different processes communicate; to the programmer, remote procedure calls were very similar to local calls. There has been a lot of debate whether or not remote procedure calls should in fact look similar or identical to local calls, given the fact that their semantics are inherently different (see section 8.3.2 for details). RPC mechanisms have to be accompanied by a mechanism to describe/encode parameter values and results (see section 8.4); sometimes this mechanism is already implicitly present in the programming language (e.g., Java RMI).

As with the message based systems that are used to implement them, RPC mechanisms suffer from possible message corruption or message loss. Since more than one message is
involved in calling and returning from a remote procedure, the absence of a reply message
does not provide a clue as to whether or not the procedure was actually executed: it might
be the invocation message that was lost, the reply message that might have gone astray,
or the process executing the procedure that might have failed. Depending on the way
such message losses are handled, RPC systems will exhibit different call semantics (see
section 8.3.2 for details).

With the transition from procedural programming to the object-oriented paradigm, the
idea of remote procedure calls gained additional interest. This lead to distributed objects
and remote method invocation (RMI). Both will be described in the next section.

8.2.4 Distributed Objects

The Object Oriented programming model is particularly well suited for remote procedure
calls. This is for reasons that are related to both the Object Oriented design methodology
and to specific Object Oriented language mechanisms. First, it is a widely accepted design
strategy to model objects to correspond to real world entities, and one reason to resort to
distributed systems is that the represented or controlled entities are themselves distributed;
distributing such systems at the object level therefore comes naturally. Second, standard
O-O design patterns allow almost all of the complexity of remote invocation to be hidden
from the programmer. Distributed objects generalize the concept of remote procedure calls
to method invocations on remote objects. The mechanism is usually referred to as remote
method invocation or RMI [CDK01]. Objects that hold a reference to a remote object do
in fact hold a reference to a local proxy for that object. This proxy implements the same
interface as the remote object; its task is to forward all method invocations to the latter,
accept the results and return them to the caller. All this is done invisibly to that caller, with
two exceptions: due to the aforementioned difference between local calls and remote calls
(section 8.3.2), the failure mode of remote method invocations is inherently different from
that of local method invocations. As we will see in section 8.4, the semantics of parameter
passing also differ between local and remote calls.

The Object Management Group (OMG) was to define the first widely available form of
distributed objects, known as CORBA (Common Request Broker Architecture). CORBA
[OMG02a] uses a single interface definition language (IDL, see section 8.4) to support clients
and servers written in any implementation language for which an IDL-compiler is available.
This makes it relatively easy to interface existing systems written in those languages to
newly developed systems.

A modernized and more flexible form of the mechanism is supplied as part of the Java Plat-
form: Java RMI. One major advantage of Java RMI over other RMI implementations like
RPC, DCE, and DCOM (see section 8.4) is its full support of remote object polymorphism.
Java RMI calls can exchange and return arbitrary Java objects, not just values of a number
of predefined types\(^2\). In the case that the object to be transferred is an instance of a class

\(^2\)Values are transferred as serialized deep copies, unless they are remote references.
that is unknown to the receiving virtual machine, the class is downloaded automatically and transparently. This automatic code transfer results in code that is dynamically portable, and it allows for systems that are dynamically extensible. The price for this flexibility is platform dependence: even though Java code can be interfaced to (e.g. legacy) systems written in different languages, such hybrid systems would lose the advantage of remote polymorphism.

Another development related to method based interaction is directed towards increased interoperability. This is achieved by using platform independent means (i.e., XML [HM01, McL01]) and a widely supported transmission protocol (HTTP) as a basis. One de-facto standard should be mentioned here: SOAP (Simple Object Access Protocol, [Eng02, McL01]). SOAP is a protocol based upon XML, and uses that language both to implement its own functionality and to encode the data to be transmitted. SOAP supports method invocation as well as messaging. In the former case, the method name and arguments are encoded as an XML document; in the latter case, the message itself is an XML document. Both method calls and messages are wrapped in an XML ‘envelope’ that carries the information needed to have them delivered. One significant advantage of using HTTP as the transport layer is the transparency of firewalls for HTTP: special measures will usually be required to allow systems to communicate if other protocols are used (hence the effort to implement RMI over HTTP, see [JXTb]). A possible disadvantage is the security of the documents as they are transmitted: XML documents are readable (though not necessarily understandable) for everyone that gets to see them. This will require special measures to encode all or part of the transmitted documents (see also [Kay03]).

8.2.5 P2P systems

A relatively recent addition to the family tree of distributed systems is formed by Peer-to-Peer (P2P) systems. P2P systems drop the distinction between clients and servers: every peer can function as both (at least in principle). Most P2P systems were initially developed as part of a particular application. Examples are Microsoft Messenger (MSN), ICQ (a system similar to MSN) [ICQ], Gnutella (a (music) file sharing system) [GNU]. While most of these pioneering applications use proprietary mechanisms, there is at least one effort underway that is directed towards the development of an open standard for P2P communication: the JXTA project [JXTa, OTG02, Wil02]. Whereas most P2P systems provide a service that is central to a group of identical peers, it is also possible for different kinds of peer to form a so-called peer group and exchange information. An example of the latter kind would be an auctioning system that consists of both seller and buyer peers. The absence of a central server and the option of replication of services over peers makes P2P systems very robust in the face of unavoidable hardware and network failures.

Communication between peers serves more than one purpose: it is used by peers to discover other peers, to query discovered peers for a description of their peer services, and to inform

---

3 Obviously, distributing the ‘server’ application introduces the risk of causing massive network congestion, so special care should be taken when designing such systems.
discovered peers of their own peer services. Last but not least, communication is used as part of the execution of those services. Usually, P2P systems use messaging to interact. In the JXTA platform, there is a specific class of messages (called advertisements) that is used to exchange information in the form of XML documents. Advertisements are heavily used by the JXTA platform itself to set up and coordinate between the peers. Messages (advertisements) are sent through pipes, which are unidirectional communication channels (bidirectional pipes might be added to the JXTA definition at some stage). A pipe is an abstraction that defines a pair of end-points that can be connected to peers. End points can be communicated between peers by means of end-point advertisements. This way a network of peers can be (re)configured to fit the needs of a particular application. E.g., the peer network might need to be reconfigured in the face of hardware or network failures. Even though much of the traffic between peers will consist of such advertisements, peers are free to use pipes to communicate other information, such as non-XML or binary messages. This will happen, e.g., to transfer files between peers. It is even possible to perform remote method invocations between peers, e.g., RMI-over-JXTA [JXTb]. The latter implementation is compatible with regular Java RMI, allowing the developer/user to use a simple switch to choose between socket based RMI and RMI-over-JXTA.

8.3 Local/Remote transparency of RPC/RMI systems

There is an important aspect of remote procedure calls and remote method invocation that has not been mentioned yet: the semantics of remote method invocation is inherently different from that of local invocation, even if the invocation syntax may look strikingly similar. In fact, the initial goal of completely hiding these inherent differences, as was done for RPC's [BN84], has been argued undesirable by others [LS82, WWWK99]. We will focus here on syntactic differences and markers (section 8.3.1), the failure model that is implicit in the call semantics of remote method invocations and the parameter passing mechanism that is different between local and remote invocations (section 8.3.2).

8.3.1 Syntactic differences

It is good practice in the field of programming language design to use different syntax for things that are semantically different. Hiding such differences makes learning a language more difficult and is a common source of programming errors. For this reason a discussion has been going on for a while on if and how to make explicit the difference between local and remote procedure calls. There are several options:

- The difference can be made explicit by means of specialized syntax [LS82]; this allows for extra facilities like aborting calls that take too much time.

- Others choose to limit the distinction to a special type marker keyword [Mey97] or to marking a class with a tagging interface [Gro01].
8.3. Local/Remote transparency of RPC/RMI systems

The current consensus is that the syntax of method invocations should be the same for local and remote calls, and that the difference should be expressed at the interface level. The justification for this choice is that the remoteness of an interface and of the methods that it exports does not change the logical correctness of a program, but that it requires the programmer to be aware of communication problems and/or partial failure (see also section 8.5). Since most RMI systems specify that remote methods must throw a remote exception, programmers are indeed forced to handle these problems at the call itself.

8.3.2 Semantic differences

The semantic differences between local and remote procedures calls are significant, and every developer should be fully aware of them. In this section we will concentrate on the differences themselves: the consequences for the design of logistic contracts will be discussed whenever appropriate. The relevant semantic differences are the failure model and the semantics of parameter passing.

Failure model

As was discussed above, RPC/RMI systems use messages to implement their higher level of communication and messaging might be unreliable, causing messages to be dropped, delivered out of order, or delivered more than once. Every RPC/RMI system has to cope with such problems, and different systems do so in different ways. The semantics of remote procedure calls may therefore vary; we refer to them using the names from [CDK01]:

maybe: With 'maybe' semantics, there is no way for the caller to find out if the remote procedure call has been executed or not, or if the server process has crashed while executing the procedure. This semantics results if there are no automatic retransmits of the call request if no reply is received within the timeout period.

at least once: With 'at least once' semantics, either a reply is received (meaning that the procedure terminated normally), or an exception is thrown if no reply was received. This semantics results if requests are retransmitted if no reply was received before the request has timed out and if requests are not checked for duplicates. Since the procedure might end up being executed more than once, unpredictable results might be obtained unless the procedure is idempotent 4.

at most once: With 'at most once' semantics a reply is received, meaning that the procedure terminated normally, or an exception is thrown indicating that no reply was received. In the latter case, the procedure may or may not have been completed. This semantics results if requests are retransmitted and filtered for duplicates, and if replies are kept and retransmitted when necessary.

---

4A procedure invocation is idempotent if the observable effects of executing it more than once are identical to the observable effects of executing it exactly once.
Both Java RMI and CORBA use 'at most once' semantics, while Sun RPC uses 'at least once'. Some languages allow the user to specify the required semantics, e.g., CORBA offers 'maybe' semantics for designated procedures. Note that with each of these possible semantics, it is still possible for a reply message to arrive after the timeout (in case of 'maybe') or after an exception was thrown (in case of the other options). Such spurious reply messages must be discarded automatically by the receiving party.

**Parameter passing**

An important aspect of each RPC/RMI system is its mechanism to encode and pass parameter and return values. In most specialized interface description languages (see section 8.4), parameters and return values are limited to the types defined by the various supported implementation languages; sometimes the IDL defines some additional types that are then mapped onto the implementation languages. In the cases where a programming language is used to describe remote objects, parameters and result values may usually be instances of arbitrary objects. In none of the RPC/RMI systems it is possible to pass pointers as arguments to remote procedures; this would only be possible in systems supporting distributed shared memory. The parameter passing mechanism is always by value for primitive and structured types, by reference for remote object references, and by deep copy for local object references (if supported). This means that the semantics of parameter passing may be inherently different for local and remote procedures or methods.

Parameter modes are often based upon the model offered by the implementation language, e.g., in Java RMI (which is pure Java) the mode is always in, whereas in CORBA IDL parameters may be designated in, out, or in out. In the latter case the individual language mappings might need to specify additional mechanisms to support non-native parameter modes (e.g., an encapsulation mechanism is used to support modes out and in out when interfacing to Java objects [OMG02b]).

### 8.4 Interface Definition Languages

One thing that is common to both RPC and RMI systems is the need to define the interfaces of remote server processes and remote objects. Sometimes these interfaces are defined using a regular programming language (e.g., Java, Eiffel); other systems require a special interface description language (IDL) to describe the signatures of remote procedures or the interfaces of distributed objects. The latter class of system is often independent of an actual programming language and can be used to create interfaces between systems written in different languages. This is an important advantage for large enterprise applications that might need to cross platform boundaries. The price of cross-platform compatibility is the loss of full remote object polymorphism and code transfer (see also section 8.2.4). Obviously, the issue of transparency as discussed in the beginning of section 8.3 is mainly relevant for systems that use regular programming languages as an IDL.
One of the earliest interface description languages is XDR [Sri95b]; its name still shows that it has evolved from a language for external data representation. XDR has a notation for defining constants, typedefs, structs, unions, enums, and unions. Parameter types are limited to primitive types and the types that can be defined using the XDR notation. An XDR 'interface' refers to exported procedures by means of a procedure number. XDR procedures may have a single argument and a single result, so structs must be used to pass more complex data. Call semantics of RPC's is at least once (see section 8.3.2). The OMG has defined an interface definition language, IDL, as well as a data representation language, DCE, as part of the CORBA architecture [OMG02a]. CORBA IDL supports the definition of modules, interfaces, types, attributes, and method signatures. CORBA interfaces allow for interface extension as known from some O-O languages. Initially, CORBA did not support the passing of local objects as parameters: it was possible to pass references to other remote objects though. As of version 3.0, CORBA has added the possibility to pass deep copies of local objects by a form of object serialization called 'Objects by Value' [OMG98, Vin98]. Due to the fact that objects might be written in different languages and/or running on different platforms, passing of behaviour is not generally possible. Exceptions to this rule exist, e.g., when the receiving object has access to some kind of runtime environment in which the downloaded behaviour can be executed. At the very best, the passing of behaviour is not as straightforward as in a single-language system. A recent OMG specification that elaborates on the 'Objects by Value' concept is the Java Language to IDL mapping [OMG03]. This specification allows Java systems to use a subset of Java RMI to map directly to Corba objects; it requires the receiving platform to have access to a class that is analogous to the class that is transmitted. CORBA IDL allows the programmer to specify the calling semantics of each method, which can be either maybe or at most once (see also section 8.3.2). The most flexible way (with respect to the availability of the full spectrum of Object Oriented techniques) to describe interfaces is provided by systems that use a regular programming language, e.g. Java RMI, Eiffel, or (more recently) C#*. These systems provided full object orientation, including the passing of behaviour, over process boundaries. In contrast to these fully object oriented systems, CORBA might better be called an object based system.

Remote procedure calls are usually supported by specialized tools to generate the code that takes care of the lower level communication. This is always the case for interfaces that are written in a dedicated IDL: such interfaces are compiled into regular code by an IDL-compiler. For languages that use a 'tagging interface' (see 8.3.1), e.g. Java RMI, a 'post-compiler' is used to generate the additional code from the compiled application code. In languages that have linguistic support for remote method invocations, e.g. Eiffel, it is the language compiler itself that can generate the required low level code.

---

*This was because the supported implementation languages did not even have to be Object Oriented.
8.5 Partial failure of distributed systems

While distributed objects provide a very natural means to implement the logistic services and actors that we have described in the remainder of this thesis, the distribution of actors over a network is not without some problems. In particular, sequential software systems or concurrent systems in which all processes reside on a single machine are inherently more reliable than software systems that involve networks of multiple machines [OW00, AOS+99]. The problems resulting from unreliable communication were already mentioned in section 8.3; in this section we will concentrate on the problem of partial failure.

Whereas on a single machine all processes making up a working system are likely to fail together (partial failure of such systems usually being the result of software bugs, a cause that we do not consider here), the opposite is true for a network of distinguished computers. In any distributed system there will be intermittent network failures, temporary disconnections, machines that fail. Some of these failures are the result of hardware or software malfunction; others may be the result of regular maintenance and/or upgrading procedures\(^6\). In almost every such case, it is only one machine that fails, or only one part of the distributed system that does not respond to some request.

Partial failure is a problem because there is no central component that can determine that a part of the system has failed and that can inform the other parts of the system. Therefore, these other parts have no way of deciding if a component just needs some more time to complete its task, if the system it runs on has become temporarily inaccessible, or if the component has failed entirely. Not only is it impossible to determine the failure of a component, it is impossible as well to determine the state of the system at the time of a failure. Therefore, components of a distributed system have to be designed in a way that allows other components to determine their state, and they have to be able to construct a reasonable state after some failure has occurred. To conclude, failure and performance should not be regarded as 'just' implementation issues. Instead, they should be reflected in the design of the component interfaces [WWWK99].

8.6 Remote Object Discovery

One aspect of remote procedure calls and remote method invocation has not yet been mentioned: how to find the process to talk to. This section will mention the most common mechanisms, i.e., naming and directory services (section 8.6.1), dynamic lookup services (section 8.6.2), and peer-to-peer peer discovery (section 8.6.3).

\(^6\)Parts of the system might need to be migrated to different hardware while the system is operational, and parts of the system itself might need to be upgraded without disturbing other parts.
8.6.1 Naming and directory services

There have been naming and directory services ever since computers got connected to a network and were able to provide services to each other. One of the most commonly known name services is the Internet DNS (Domain Name System) [Moc87a, Moc87b], which maps symbolic Internet host/domain names to the IP numbers they represent. The DNS is a hierarchical distributed system: there is no single database containing all mappings. Instead, the system uses a combination of centralized information to represent the top level and distributed information for the various domains and sub-domains. The stored names themselves are hierarchical too, and consist of a sequence of simple names that are separated by dots. The hierarchy of the name itself represents the hierarchy of domains and sub-domains, and thus (to a certain level) the physical hierarchy of the underlying network. Two widespread naming schemes that are based upon DNS host names are e-mail addresses and URLs (Universal Resource Locators). The DNS system can also be used to resolve host names for the purpose of inter-process communication. The host name is then used to obtain an IP number, which is combined with a port number to allow a process to connect to the given port at the addressed host. In order to achieve acceptable performance, the DNS system caches resolved IP numbers at intermediate nodes. Because cache entries are not automatically invalidated when a host goes down or gets disconnected from the network, cache entries may be stale. As entries are removed from the cache after some period, such stale references will die out eventually.

Most traditional naming and directory services look up names (i.e., strings of characters) and map them to network locations. Most of them support hierarchical names, as in DNS. Such naming schemes still require applications to know the network locations of any other applications they need to communicate with: they merely allow them to use symbolic locations instead of physical ones. More elaborate lookup would be possible by encoding relevant information in the form of strings and using some form of pattern matching. Such encoding schemes would allow prospective clients to use a service description rather than a service location to find any instances of applications providing the service. Naming schemes like these are sometimes referred to as intentional naming systems. The next system will briefly describe such systems.

8.6.2 Intentional naming

A recent naming service that has taken a step along the path toward intentional naming is the Lightweight Directory Access Protocol (LDAP) [WHK97, HM02]. LDAP uses attribute-value pairs to specify a path through a hierarchical name structure. This path, which LDAP refers to as the Distinguished Name (DN), uniquely identifies a service independently from its location.

A more extensive proposal for intentional naming is provided by INS (Intentional Naming System) [AWE99]. INS also uses an attribute-value sequence to identify services, but it adds some extra features to the system: while names uniquely identify a service, that
Chapter 8. Distributed computing

does not necessarily mean they identify a unique provider of that service. INS allows
for (e.g. performance or availability) metrics to be used to choose between the possible
providers of the service. This makes it possible to implement some kind of load balancing
scheme that depends on actual provider load or network delays. INS uses INR’s (Intentional
Name Resolver) to route service requests to providers; INR’s exchange information between
each other in order to disseminate knowledge of available services and the routes to the
corresponding providers. Another new feature of INS that will become more and more
important in future systems is support for mobile applications.

Whereas attribute-value pairs are a huge step forward compared to ordinary or hierarchical
names, the system does not offer guarantees that the name provides an accurate description
of the service offered. Using an interface describing the service in more detail is a solution
that comes much closer to that ideal. The Jini lookup service is an example of this scheme:
it allows services to register or unregister their service interface (i.e., the Java interface that
is implemented by the service provider) with the lookup server dynamically. Providers are
selected on the basis on their interface and possibly some additional service attributes, like
their location. New lookup server instances will make their presence known (by broadcasting
a message) so running processes can also register with those new lookup servers. The
Sun reference implementation of the Jini lookup service does not exchange information
between instances. A lookup server usually requires processes to maintain (lease) their
registration; if the process dies or becomes unreachable, the lease will expire after some
time and the reference will be removed. Whereas the Jini lookup service is language-
centric, the same mechanism can be used in conjunction with services that are described by
a language-independent IDL interface (e.g., CORBA-IDL).

A intentional scheme like described here is particularly well suited to maintain information
about devices that are attached to some local network, like scanners, printers, faxes,
camera’s, etc. If appropriate leasing and/or aging mechanisms are used, the scheme is very
useful in an agile environment. The scope of lookup services is usually equal to the scope of
the network broadcasts used to locate them. If would be possible of course to create groups
of lookup servers that know about each other’s network location and that share information,
but this is not a feature of the Jini lookup service. Most current naming systems can
be scaled up to a certain extent, but do not scale well enough to cover the entire Internet.
See [AWEBL99] for more information.

8.6.3 Peer discovery

Whereas the interaction aspects of P2P were covered in section 8.2.5, we will concentrate
now on the discovery service that is an intrinsic part of P2P systems. Since JXTA is
intended as a generic platform, we will focus on that system.

One of the goals of P2P systems is to build reliable and scalable systems that are centered
around a particular service. Most services are application specific, but some services are
defined by the platform itself. One of these core services is the discovery service, that allows
peers to locate other peers in a given peer group. The JXTA platform defines rendezvous peers and router peers, that allow peers to discover other peers across firewalls and NAT (Network Address Translation) systems and to access peers that cannot be reached through a fixed route. Rendezvous peers are necessary because a peer can only directly discover peers that are on the same local network. To discover peers that cannot be reached by a broadcast, a peer can contact one or more rendezvous peers of which it knows the address (called static discovery), and download all peers the rendezvous peer knows about. This set may contain other rendezvous peers that have been discovered dynamically. This mechanism allows peers to find all peers that belong to the same peer group\(^7\). Firewalls can be traversed by contacting a proxy peer that acts as a proxy for a rendezvous peer that is at the other side of the firewall; this proxy uses a network protocol that is allowed through the firewall, usually HTTP. Routing peers are used by the peer group to map unique peer id’s onto the routing information that is needed to communicate to the peers. Routing peers are flexible enough to provide new routing information even when a route has changed when a message is already halfway its journey.

### 8.7 The preferred platform for LinC

Now that the preceding sections presented an overview of some the most important aspects of distributed systems and the techniques to implement them, this section will focus on the selection of a software base for the LinC platform. We will start with the two main choices to be made: message vs. method based interaction (section 8.7.1), and the language centric vs. the language independent approach (section 8.7.2). The last section (section 8.7.3) will select the lookup/discovery mechanism that has the best support for agility.

#### 8.7.1 Messages and/or methods?

One of the main goals of the LinC platform is to apply the idea of ‘design by contract’ to the design of logistic systems and actors. While it has been clear from the very start that LinC was to be based upon the distributed object paradigm, there still was freedom of choice with regard to the interaction mechanism: in principle, actors could use asynchronous message based interaction, asynchronous or synchronous interaction through a shared data space, or synchronous method based interaction. Earlier publications about logistic contracts were not very explicit with regard to the style of interaction: while they defined a message-like mechanism (called Briefs) [ELL00, Eve00], they combined this basically asynchronous mechanism with a protocol-like script that represented an inherently synchronous mechanism [ELL00]. In a more recent publication about TrAcEs and FoRcEs, Lindeijer [Lin03] has argued that those systems should support both message based interaction in the form of ‘Letters’, ‘Signs’ and ‘Signals’, and method based interaction using

\(^7\) Since peers, and rendezvous peers in particular, often rely on cached information about peer locations, it is possible for discovered peer references to be stale.
'Methods' as well as native Java RMI based interaction. Each of these mechanisms was to support remote polymorphism, though, including the transparent downloading of classes. Since LiNC is intended to complement TrAcEs and FoReEs (see also section 2.1), one might expect the LiNC platform to also support all of these interaction mechanisms.

Looking at the LiNC language as specified in part II, it will be clear that LiNC contracts are mostly about specifying ordering and concurrency constraints to the tasks (messages) that can be executed (handled) by the actors involved, and that such constraints represent a certain bias towards a tighter form of control than is possible with asynchronous mechanisms. In particular, the possibility to specify concurrent execution of tasks assumes that these tasks take time to complete\(^6\). The reason for this linguistic bias towards method based interaction is that LiNC contracts are meant to specify the entire range of interactions that are possible in logistic systems, i.e., from the lowest level of machine control through the middle level of traffic control up to the highest level of logistic control (see also section 2.1). At the lower levels, the dominant interaction method is surely the method call; this is because methods are easier to work with and because the lower levels of logistic systems are more likely to benefit from the flexibility of remote polymorphism and downloadable code than the higher levels. Also, if message based interaction is needed, 'messenger methods' can be used. Such methods do not have the advantages of high-level message based systems though: in particular they will not be queued, and the receiving party has to be online during the entire contract execution. Using messenger methods, it is easier to meet real time requirements than with asynchronous messaging systems. It is for these reasons that our language had to support at least method based interaction, and that LiNC *tasks* are equated to *methods* in the implementation language.

At the higher levels of logistic systems, though, the very concept of 'real time' has a different meaning, and is not necessarily incompatible with asynchronous messaging. At the same time, network latencies and temporary interruptions are more likely to occur at those levels. Also, the spontaneous grouping of actors to form ad-hoc logistic enterprises and the reliability and verifiability of these operations are often more important than the tighter control of synchronous operation. Obviously, it would not be too difficult to integrate asynchronous messaging into the LiNC framework and language. One would have to account for the fact that messages don't take time to complete and that their use in concurrent constructs requires some additional rules. To allow the compiler to check these rules, it would be necessary to complement the *tasks* sections of a LiNC contract with a set of *messages* sections. The registration of actors as listeners for these messages could be hidden from users and performed by the basic actor class from which specific actors have to be derived (see also section 9.3). By extending the language with messages, the trade-off between the benefits of a relatively loose coupling offered by message based interaction and those of the much tighter coupling offered by method based interaction can be made by the designer of logistic contracts, instead of being forced upon him by the language.

\(^6\)This does not rule out the possibility of defining tasks that do not perform any computation and behave just like messages, of course, but such 'messages' are still sent in a synchronous manner, thereby blocking the sender until control has returned from the receiver. The interaction between sender and receiver is therefore fully synchronous.
8.7.2 Language-centric or language independent?

Having selected a particular communication model, the next choice to make is that of a suitable software base to build upon. Important factors in this decision are the advantages offered by true remote polymorphism as offered by a language-centric system like Java RMI, and the ease of incorporating existing software components and crossing platform boundaries as offered by a language independent systems like CORBA and SOAP. Traditionally, these two options have been mutually exclusive, but this is changing; the JVM is available for many common platforms and Java allows for easy integration of existing components using the Java Native Interface (JNI). Crossing platform boundaries is therefore not a serious obstacle any more (at least not for this particular language-based platform). At the same time, the language independent CORBA platform is integrating the passing of objects by value (see section 8.4), thus accommodating a primitive form of remote polymorphism. The other language independent models do not offer remote polymorphism though. Because of these limitations the language-centric solution, despite of its own remaining limitations, was the preferred choice for the implementation of TrAcEs and FoRcEs (where code downloading is a significant advantage [Lin03]). The same solution was therefore adopted in the implementation of the LINC platform. It is foreseen that future implementations of the LINC platform will have increased support for interoperability through the addition of a language independent interaction mechanism (probably a solution that is based upon XML and/or SOAP or their offspring).

8.7.3 Support for agile systems

One of the main reasons for developing the LINC platform is the need to support temporary actor networks. Logistic contracts not only serve as the 'glue' that connects those actors, but also as a template to locate suitable actors in the first place. As actors may come and go and the set of services they offer may vary over time, the way to locate actors and services is a crucial part of the platform. The same mechanism that is to cope with agile actors can be used to handle (transient) network failures.

The lookup mechanism that looks the most promising on the short term is Jini-based discovery. The main reason for this is the fact that Jini allows actors to search for parties that implement a particular interface, optionally supplemented by additional information. This makes it very easy indeed to search for parties that play a role in a particular service contract. Since parties have to lease their 'advertisement' from the Jini lookup service, the advertisement would be canceled automatically if the party should have stopped business, crashed or just become unreachable. It is for similar reasons that Jini was used in the implementation of the other AgileFrames components, TrAcEs and FoRcEs [Lin03].

The main disadvantage of the Jini lookup service is that it must either be reachable through a (low-level) network broadcast, or reside at a known network address/port number. This means that an actor can only find parties that have registered with the same (set of) lookup services as the actor itself. While this might be an advantage for services that have only local
validity, it will be a definite disadvantage for services that serve a much wider community. Since the research presented in this thesis focused on interaction between actors rather than actor discovery, those disadvantages were relatively unimportant.

Further research has to determine what is the best mechanism for actor lookup in the context of logistics systems. The requirements are likely to include intentional naming, interface-based lookup, ability to discover actors that are behind firewalls and NAT systems, mobile actors, and leasing of advertisements. One way to combine the advantages of Jini-based lookup with those of P2P-based lookup is to use a Jini lookup service that is a member of a JXTA peer group containing other (Jini) lookup services. Bridging to other interface based lookup services would also be possible.
Chapter 9

The LinC platform architecture

This chapter offers an introduction into the architecture of the LinC platform. The main goal here is to show that the ideas presented in the previous parts of this thesis can indeed be implemented in a form of a system that allows contracts to be validated in real time. As a positive side-effect, it shows what form such an implementation could take and thereby provides some guidelines for implementors of the platform and its supporting tools. It is assumed that readers have some basic knowledge of both Java RMI and Jini (readers that are new to RMI and/or Jini are advised to read appendix A first). Section 9.1 will give some important details on the Logistic Contract Space, such as the all-important naming conventions, the way in which the Jini lookup service is used, and the life cycle of logistic contracts. Following that, section 9.2 will describe in some detail how the parties that cooperate in the execution of a contract will be instantiated, how the protocol is verified, and how the results of execution are used to build a journal tree. The next section will define the requirements that LinC actors have to fulfill; it will also describe some capabilities that are shared by every LinC actor (section 9.3). Finally, section 9.4 will define some of the tools that would be part of a practical LinC environment.

9.1 The Logistic Contract Space

Section 3.4 introduced the Logistic Contract Space as a structured name space where contracts are stored; section 5.6 provided some additional detail by introducing domain names and service names, respectively. In this section we will see that there is no single physical location where contracts and actors are stored: there is no such thing as a centralized database containing contracts or service actor implementations. Instead, the Logistic Contract Space will be distributed over many different systems sharing a common organizational principle allowing it to be seen as a logical whole. This organizational principle consists of a naming scheme and a formalized community process [JAV, JIN] that decides on domain and service names. The community serves as a central authority that enforces an organizing principle upon its members, thereby following the principle of voluntarily subjecting oneself to a central authority in order to guarantee sustained usability.
In order to be accepted by prospective users, it is important that all contracts and actors stored in the Logistic Contract Space can indeed be trusted. For contracts, this is trivial: every contract just states what it is (does), and can be used on a *take it or leave it* basis. For actors, matters are different, as they must be trusted to offer the service they pretend to offer: there must be a strong link between an actor and the textual form (!) of the contract(s) it may service, and any tampering with either the actor or the contract must invalidate this link. A mechanism to accomplish this has not been defined yet, but doing so is not expected to pose any major problems.

Figure 9.1 shows a very simple Logistic Contract Space containing a small number of domains that are relevant for transportation. Though conceptually one, the Logistic Contract Space and the (sub-)domains contained therein consist of the union of any number of partial instances of (sub-)domains and their contents (i.e., services and further sub-domains).

![Diagram](image)

**Figure 9.1: Example Logistic Contract Space**

These services and sub-domains may be scattered over a large number of machines on a network; each of these machines may contribute services and/or sub-domains to any given (sub-)domain of the Logistic Contract Space. Referring to figure 9.1, the sub-domain *ParcelTransport* might reside on a single machine called A; sub-domain *PalletTransport* might reside on another machine called B; and finally, sub-domain *ContainerTransport* might be scattered over 3 different machines, A, B, and C. As a consequence, the parent domain (i.e., *Transportation*), will also be scattered over these three machines. Each of the machines that contribute to the same (sub-)domain makes its part of that (sub-)domain accessible to external parties (by means of the Jini Lookup Service). As a result of the distributed organization of the Logistic Contract Space, service domain names need to be standardized but do not have to correspond to unique physical locations. This is also true for service names: even though any service domain can contain at most one service with a certain name, the same service can in principle be present at many different locations.
Obviously, all instances of the same service contract at different machines have to be identical\(^1\), but the requirement that any and all contracts that go under the same name are to be identical is difficult to fulfill without forcing the name authority to not only authorize contract names, but also their actual contents. In particular, no copy of a service contract or logistic actor could be made without the explicit permission of the central authority. This would be a significant extension of the role of such an authority, which was initially only acting as a clearing house for domain (and possibly service) names.

Sofar, we didn’t mention the aspect of visibility and access rights. It is not clear yet if there is a need to limit visibility and access to the contract texts that are stored in the Logistic Contract Space. The same holds for the corresponding code and the actor code that has been uploaded to the Logistic Contract Space. Since both contracts and actors must be downloaded through e.g., HTTP, any individual server could decide to restrict access to registered parties only. In addition, actor code could require parties that want to engage in a contract to login before they are accepted.

### 9.1.1 Naming conventions

The naming convention used for Java packages can be applied to service contracts as well: the domain name and service name specified for a contract can be mapped to a package name. That package will then contain the classes and interfaces needed to implement the service actors as well as any supporting classes and interfaces (more about this in 9.2.1 and 9.2.2). Following the Java conventions, we have adopted a naming scheme in which the package name corresponds to the reversed Internet domain name of the service provider, followed by ‘linc’, followed by the specific LINC domain name. E.g., inventing some specific provider names, the service domains given in figure 9.1 would correspond to the following package names:

```plaintext
com.evergreen.linc.Transportation.ContainerTransport
com.danzas.linc.Transportation.ContainerTransport
com.tnt.linc.Transportation.PalletTransport
com.dhl.linc.Transportation.ParcelTransport
```

### 9.1.2 Using the lookup service

The Jini Lookup Service provides a flexible and reliable means to locate service contracts and providers, and the Jini leasing mechanism is appropriate to adapt to changeable provider and client networks. As we will see in section 9.2.1, not only service providers may register with the lookup service: in the LINC framework, clients may do the same. Even though client and providers both register for the same service, they can only do so for their own specific role in the contract: so while providers are waiting for clients wanting to obtain their services, clients will be waiting for providers that become available to

\(^1\)To allow contracts to evolve over time, a versioning mechanism should be used.
fulfill their needs. This symmetry may be extended to cover multi-party services, allowing
the group of parties to be extended until it is complete after which the contract can be
instantiated. The current prototype implementation is for 2-party contracts only.

The Jini infrastructure requires parties to register themselves at the Lookup Service. There
may be many instances of this service across the network, so reliability can be improved by
registering at all accessible lookup services. By doing so, chances of a network disruption
that prevents parties from locating each other are greatly diminished; this is especially
important because the LiNC framework spans a large geographic area where such disruptions
may be quite frequent. Registered services are organized in so-called groups; these groups
can be used to distinguish between e.g., application domains and/or physical location of
the provided services. In the LiNC implementation, these lookup groups are used to im-
plement service domains. Since any group can contain subgroups (and so on, recursively),
there is a direct mapping between service domains and groups in the Jini Lookup Service.
The special public domain is mapped upon the default group, while the special private
domain is mapped upon a group with some reserved name that can only be accessed locally.

9.1.3 Fully and partially implemented services

The Logistic Contract Space is designed to contain both fully implemented services and
services that are only partially implemented. When a service is fully implemented, the
Logistic Contract Space would (at least) contain all of the following:

- the text of the service contract,
- the classes and interfaces that resulted from compiling the contract,
- the actor specific classes that implement the desired behaviour for all the actors in
  the contract.

When all of these are present, any party can download and execute the code of any actor!
Obviously, this is usually not the case. Assuming that the Logistic Contract Space will
always contain the service contract itself and the classes and interfaces that resulted from
compiling it, it is more realistic to find oneself in a situation where some actor implemen-
tations are in the Logistic Contract Space, while others are not; in this case we will call the
service partially implemented. A party could then download and execute the actors whose
code is available, and try to locate the actors whose code is not available by means of a
discovery service (see section 9.1.2). This situation will typically arise when some actors are
relatively simple and standard and can operate in a stand-alone fashion (e.g., like applets
that do not rely on any other software at the client side), while the remaining actors cor-
respond to larger scale applications representing an actor’s business processes. The latter
parties can only be contacted if they have registered themselves as an actor of the service
in question. Of course, the party that wanted to execute the contract in the first place can
also register itself as an actor of the service (or, less likely) upload its own implementation
to the Logistic Contract Space.
9.1.4 Abstract versus concrete contracts

Given the above, it would be nice for a party to know if a contract can be executed without the need to first develop one or more actor implementations. As we have seen above, this is the case if all parties are either available for download or can be contacted through a discovery service. We call such contracts concrete. If one or more parties implementations are still missing, i.e., if they can neither be downloaded nor contacted, we call the contract abstract. It is impossible for any party to engage in a contract unless that contract is concrete or will become so after the party has registered itself or has uploaded its implementation code.

Instantiation of a service contract will now proceed as follows:

- a contract is selected,
- the code for all actors whose implementation can be found in the Logistic Contract Space would be downloaded and executed,
- the interested party registers its own actor implementation, if any,
- the registered parties that are to become the remaining actors are selected.

After all these steps have completed, execution of the contract will start without further actions on any party’s side. All registered parties will be contacted automatically. Note that we did not make a distinction between clients and providers here: any contract party (except the downloadable ones, of course) can be the initiator of the process! Of course, the Logistic Contract Space will have to be supported by the appropriate tools to search for contracts, download ready-made implementations, and contact registered actors. Section 9.4 will make a start exploring such tools.

9.2 Contract execution

Without going into too much detail, we will now explain the main technical aspects of contract ‘execution’. As we already explained in section 3.3.3, contracts are not executed in the usual sense of the word; instead they are used to verify the interaction between the actors involved. Before any such interaction can take place these actors have to be instantiated, and before they can be instantiated, the actors must have discovered each other. They do so by advertising themselves in the Jini lookup service (see also section A.2.1) and by querying the lookup service. Since the LN/C platform is symmetrical, it may be the provider who advertises itself as the provider of a certain service, or the client who advertises itself as the client of a service. The following sections will describe the different stages of a contract’s execution. We will start with contract instantiation.
9.2.1 Contract instantiation

The left side of figure 9.2 depicts the client side of a 2-party interaction, while the right side of the figure depicts the provider side. Both sides have a so-called starter and a so-called registrar. If a party wants to advertise that it is available to fulfill a particular role in a contract, it has to instantiate a registrar for that role, i.e., a ServiceProviderRegistrar or a ServiceClientRegistrar. If a party wants to find other parties that have already registered for certain roles, it has to instantiate a starter for its own role, i.e., a ServiceProviderStarter or a ServiceClientStarter. Whereas registrars register the availability of a service provider or client, starters use the lookup service to locate the complementary registrars. E.g., in case of a 2-party service contract, the ServiceClientStarter would try to locate a ServiceProviderRegistrar, while the ServiceProviderStarter would try to locate the ServiceClientRegistrar. The mechanism is very similar to what is described in section A.2.2 and in fact is implemented right on top of it: what we call the ServiceProviderRegistrar here, is the ServiceProxyImplementation of figure A.2 and the RemoteServiceImplementation of figure A.3. The same is true of the ServiceClientRegistrar. The client side hasn’t been mentioned before, but again the starter is very similar to a regular Jini client. The lookup process is illustrated by figure 9.3, which is indeed a symmetrical version of figure A.1. Whereas all parties are trusted in the current pilot-implementation, an authentication procedure might be required by starters and registrars to verify parties before the contract is instantiated (see also section 9.3).

When the service provider and client have successfully contacted each other, the next step of
contract instantiation takes place: at the provider side, an instance of the `ServiceProvider` class is created by the provider starter or the provider registrar, while at the client side an instance of the `ServiceClient` class is created by the client starter or the client registrar. These classes are renamed versions of the `RemoteServiceRegistrationImplementation` class of figure A.3 (it didn’t exist in figure A.2). The use of contract instance specific instances of these classes follows the alternative design pattern outlined in the last paragraph of section A.2.2. The actual execution of the contract is thus delegated to the `ServiceProvider` and `ServiceClient` instances, which know how to contact each other as they are given a handle of the opposite party at the time of their instantiation. The following section will explain how the contract parties can communicate.

One issue that needs to be mentioned is the instantiation of service objects, service parameters, and service indexes. An obvious question is where these values come from, and which party is able or allowed to supply them. This is especially relevant for the service objects, which act as a kind of ‘database’ representing the state of the objects or goods manipulated by the contract. The proposed solution is to allow the registrar to specify a value for each service object when it is instantiated. Thus, instantiation of a registrar leaves some service objects _bound_, while some other service objects might remain _unbound_ (see also section 5.11). Any subsequent registrars or starters for the same service are allowed to provide a value for any service objects that are still unbound; these will then become bound too. If a subsequent registrar or starter tries to supply a value for a service object that is already bound, the new value must be identical to the value to which the object is bound; since service objects are remote objects, this effectively means that the new service object must be exactly the same remote reference as the existing service object (because the two values are both remote references). If the new value is not the same as the existing
value, the new registrar or starter is not permitted to become a party in the contract. If there are any service objects that are still unbound after all actors have joined the contract, a test is performed if these service objects have a default initialization (section 5.8). If so, the service is started; if not, the service is not instantiated and the last party which joined will receive an exception. Note that the entire process is hidden inside the registrars and starters; implementors of the actors just pass the relevant remote object references when instantiating the registrars and starters for the service. The mechanism for passing service parameters and service indexes is the same, provided that these are not required to be remote objects.

9.2.2 Protocol verification

When the ServiceProvider and the ServiceClient classes are instantiated, they will get a reference to the opposite party. However, this is not just a remote reference to that party itself. Instead, each party gets a reference to a proxy instance. Because the interaction is symmetrical, there are in fact two different proxies, one for the provider and one for the client. As before, the proxies are serialized instances of the corresponding proxy classes. Whereas in figure A.3 the proxy merely hides the Jini related specifics of the communication, it now has an additional role: verification of the protocol. As every task invocation is passed to a proxy, the latter can easily check if the invocation is valid given the state of the protocol. This of course includes a check on the arguments that are passed in the call, as well as checks of the pre- and postconditions of the task. Since both parties may invoke tasks on the opposite party, both proxies need to interact with a central protocol verifier (see figure 9.2). This protocol verifier contains a suitable representation of the protocol part of the service contract. It also contains or has access to the service objects, parameters, indexes, and temporary variables. Note that in order to protect the communication between actors, the proxies might want to encrypt all data being passed to the opposite party. Encryption keys can be exchanged as part of the actors’ authentication procedure (see also section 9.3).

As the interaction between parties takes place by method calls (as opposed to message exchange), the protocol verifier has to update its internal state both when a method is first invoked and when the method has terminated. Of course, the state will only be changed if the method invocation or return is valid with respect to the current state; if not, a ProtocolViolation is thrown to the offending party, i.e., the caller if the invocation is invalid, or the callee if the return is invalid. Preconditions are checked as part of the method invocation; if they fail, a PreconditionViolation will be thrown to the caller. Postconditions are checked as part of the return; if they fail, a PostconditionViolation will be thrown at the callee.

Obviously, some special mechanism is needed to implement the validation of method return and postconditions: if the protocol verifier were to throw the exception just like that, it would be the caller who would be punished for a crime it didn’t commit! This effect is illustrated in the upper half of figure 9.4. The fat arrows in this figure denote the caller’s
thread of execution. The thread has to pass both an entry test and an exit test (indicated by the two vertical line pairs) in the protocol verifier (the box in the center). If the test fails, an exception will be thrown (denoted by a dashed line). What is needed is a way to achieve the situation in the bottom half of figure 9.4, where the callee has to catch and handle the failing exit test. After the callee has fixed the situation, it can make another attempt to return. The most elegant solution to this problem would consist of a `return` statement that can throw an exception. A task would then end as follows:

```java
try {
    return;
} 
when (ProtocolViolation c) {
    // handle protocol violation and retry
} 
when (PostconditionViolation c) {
    // handle postcondition violation and retry
}
```

However, we do not know of any language that allows a `return` to fail. In the case of Java, this mechanism would necessitate a minor modification to language as well as a change of the Java Virtual Machine. If the Java language would ever be extended to support pre- and postcondition checking, such a change would have to be made anyway.

In the absence of a solution like the one just shown, we have to adopt a solution based upon the available mechanisms. One possible solution is a 2-stage approach, in which the callee has to ask the protocol verifier for permission to return. This work-around is depicted in figure 9.5. If the return request is denied, the protocol verifier will throw an exception which must then be handled by the callee; the exception is denoted by the second dashed line in the figure. If permission to return is granted, the callee has to return to the caller immediately.
following the request. Unfortunately there is no way to enforce this requirement, so parties
must be well behaved with respect to this. It is possible, of course, to catch parties that try
to avoid the exit test, but again it is only the caller that can be notified; this exception is
denoted by the third dashed line. It is for this reason that the protocol verifier will throw
a ProcessError exception if the callee has skipped the test: a ProcessError indicates an
unknown but severe problem at the side of the party that causes it to be thrown. As such
it will be treated as a serious breach of the contract. The same exception will be thrown if
the postcondition is no longer met, i.e., if the provider really messed up during the period
between the exit-request and the actual return.

9.2.3 Contract termination

A possible problem with contracts is inappropriate termination. When a LINC contract
is designed, it is assumed that the contract will be executed until its end, and that no
parties will quit before the entire protocol has terminated. Parties that drop out, be it
intentionally or the result of some network problem, are expected to come back later and
complete the interaction. If they don’t, they are at fault. Of course the parties have to
agree on a reasonable time to complete their contractual obligations; this is not part of the
LINC language though, and will probably be a parameter of service contract instantiation.

Even though the current system does not yet use transactions in any way, termination of a
contract uses a transaction-like mechanism: after the last interaction has taken place, the
protocol manager queries all the contract parties for a ‘discharge’. Only after it has received
a positive reply from all parties, it will grant the parties permission to quit. As a result of
this (hidden) protocol, all contract parties have to stay alive until the last interaction has
successfully ended. Any party dropping out before it has been granted permission to do so
is in error and may have to face the consequences.

9.2.4 Execution journaling

A major advantage of having a protocol that defines the communication between actors, is
that the protocol defines a structure that serves as a context for the individual communi-
9.2. Contract execution

![Journal Tree]

Figure 9.6: A structured journal tree

cation acts. This protocol does not only act like a guardian that prevents possible 'noise' (like erroneous interaction attempts) from disturbing the communication, it also allows us to build a structured log of every successful interaction that took place between the actors in a contract. Whereas conventional logs are often little more than a time-stamped list of more or less detailed interactions, the LINC compiler generates the necessary code to build a tree-structured log that directly reflects the tree-like structure of the protocol. Note that the LINC language allows contract writers to specify the level of detail in the journal tree as part of the contract (see section 5.14). In particular, it is possible to specify the amount of structure that is retained, so the journal can be as detailed as a full tree or as flat as a conventional sequential log. Figures 9.6 and 9.7 show these 2 extremes: a fully structured log containing a tree node for every protocol construct, and a simple sequential log containing little more than the individual interactions. While in figure 9.7 the interactions are ordered sequentially on the basis of their time stamps, in figure 9.6 they are grouped on the basis of their contents, highlighting the semantic structure of the dialogue. Both journal trees have been generated by running the same input through systems generated from contracts that were identical except for the journal clause (see also section 5.14).

The journal tree is built and maintained by the protocol verifier\(^2\), which already needs to keep track of all interactions anyway, and which has access to the internal representation of the protocol. Even though the tree itself is local to the protocol verifier, serialized copies

\(^2\)In the prototype, the journal tree is represented by a Swing JTree.
of it can be requested by every contract party that is interested. Also, contract parties can subscribe to journal tree events in order to be notified of any changes in the tree. As we will see in section 9.4.5, it is not too difficult to design and implement a simple tree browser; in fact, the journal tree pictures that were shown above have been obtained by such a tool. This prototype implementation allows each actor to display the journal tree as a Java Swing component. Although not yet implemented, it would be very easy indeed to generate an XML representation of the tree. This XML representation could in turn be used to generate reports and allow audits. In addition, the XML representation of the journal tree can be used as input for other applications, e.g., billing, statistics, quality control, and maybe even legal services (i.e., if the contract has been violated).

9.3 Basic LinC actor capabilities

All actors that are to play a role in a LinC contract need to have some basic capabilities. These capabilities are part of the LinC platform itself, which is why we will introduce them shortly here. At the very least, LinC actors have to be able to access the other parties, which means that they must hold a reference to a proxy for each party; those proxies, in turn, hold remote references to the parties they represent (see also section 9.2.1). All actors must also be able to confirm contract termination (see section 9.2.3). Finally, all actors must also be able to handle journal tree events (see section 9.2.4). A basic implementation of all these functions is supplied by the LincBasicActor class, which every contract party has to subclass. Both the starter and the registrar (see section 9.2.1) use facilities offered by the predefined LincBasicActor instance.
Many other functions might be added to the basic actors defined by the platform; they are not part of the pilot implementation though. Examples of useful facilities are: a mechanism to retrieve information from the journal tree (section 9.3.1), a mechanism to survive crashes and resume operation at the last stable state of the contract parties (section 9.3.2), replication of actors to cope with hardware or network failures (section 9.3.3), and an authentication mechanism to validate contract parties (section 9.3.4). Other features could be added to this list, of course.

9.3.1 Ability to inspect the journal tree

Journal trees are maintained by the protocol verifier (see also section 9.2.2), and by default logistic actors receive a change event whenever the journal tree is updated. The change event contains a serialized copy of the new journal tree, so each actor is able to maintain an up-to-date representation of the dialogue. Actors can use the standard journal tree API to access the tree internally, or may implement their own methods to access, display, or print the journal tree, or to derive progress reports from them. Such methods might be accessible to other object instances at the actor's side, in particular to methods that can be started from the actor's user interface.

Accessing the tree by means of the standard Swing API does not use any knowledge about the way journal tree data is represented. A better API would use knowledge about the various node kinds to access the data in a more straightforward way. Also, to reduce overhead when a party is not interested to receive journal tree events, the basic actor might export methods to cancel the default subscription to such events and/or to re-subscribe.

9.3.2 Ability to survive crashes

To make the interaction more robust, actors should allow for a 'hot restart' in case of hard- or software failures. Before discussing possible ways to overcome such failures, we have to distinguish between failures that occur while a task is being executed (including those that occur during the passing the request or results) and failures that occur in between task executions. When a task is being executed, hard- or software failure will result in a RemoteException being thrown at the party that invoked the task; if a number of tasks runs concurrently, each of them could end with this exception\(^3\). If a hardware or network failure occurs at a time when no interaction is taking place, things are a little easier: the journal tree is stable, without any task nodes that are in an active state (see section 9.4.5). Unless one of the parties attempts to send a new task invocation message act while the failure is still there, no RemoteException will be thrown. In either case, the basic actor must have a way to reconnect to the other parties after the connection has been restored or a party has been restarted following a crash. The Jini service ID (see [OW00]) is expected to play a role here.

\(^3\)Since the RMI software itself has already performed a number of retries, the RemoteException is an indication of a persistent error.
Unfortunately, in case of a RemoteException, it is not possible to distinguish between a network failure and a total actor failure: even if the journal tree has recorded the task as having started, this does not necessarily mean that the party that was to execute the task has in fact begun its execution. And if the task was indeed started, we do not know how far its execution had progressed when the process crashed, and/or if the party’s state has already been changed as a result of the task’s execution. Uncertainties like these make it difficult to provide a standard recipe to handle remote exceptions. In some cases, it might be appropriate to continue retrying at the actor level, whereas in other cases the only thing to do might be to cancel the entire contract. Actor replication (see below) could of course be used to alleviate this problem.

It is the responsibility of an actor to save any relevant state information so that any task can be restarted if necessary. Most likely a transaction mechanism would be used by the actor to accomplish this\(^4\). When a transaction mechanism is used to ensure that every task is executed as a whole or not at all, the journal tree should be changed to reflect the completion or failure of the task. If this is done, the journal tree becomes transactionally sound: it will always be a perfect reflection of the state of execution of all contract parties. In fact, it would allow for the implementation of exactly once semantics for task invocations (see section 8.3.2). In addition, if such a journal tree is permanent, it can be used by crashed actors to recreate the state represented by the journal tree, i.e., just before the crash.

### 9.3.3 Actor replication

Until now, little attention has been paid to the possibility of actor replication, even though a replication mechanism could greatly enhance the reliability of the system as a whole. The various naming services (be it the Jini lookup service, JXTA peer lookup, or similar mechanisms) could in principle be used to make such replication invisible to users.

Using replication, contract parties would be represented by a cluster of actors, running on different machines. Each interaction request would be sent to each individual member of the cluster, and be executed by every one of them in one concerted operation. RemoteExceptions could be ignored except when all members of the cluster fail to respond simultaneously. Obviously, cluster members have to synchronize their state, possibly with the help of the journal tree. See [CDK01] for an introduction to replication. Actor replication is a active area of research in both the Java RMI community and the CORBA community [NMMS00, Mon99, MH01, Jgr]; [OTG02, Wil02] contain some remarks on replication of peers in peer groups. The LincBasicActor class could be used to hide most replication-related aspects of interaction, but probably not all: it is up to the logistic actors themselves to implement the required synchronization and recovery mechanisms. Since the focus of the current research was mostly on the feasibility and use of logistic contracts, actor replication has not been considered for the pilot implementation.

\(^4\)This use of transactions should not be confused with transactions at the protocol level.
9.3.4 Actor authentication

For contracts intended to be used in real logistic operations, there is a strong need for being able to verify the identity of parties offering and using services. An authentication mechanism must therefore become part of the framework. Both starters and registrars must require the user that launches them to identify himself and to pass their credentials to the lookup service when trying to contact the other parties. Such a mechanism obviously requires some 'central' server to exchange and validate actor credentials. A remote reference to this server, which must itself also be trusted of course, can be passed as part of the credentials. The entire procedure can be handled by the basic actor, so the actor specific code does not need to bother with it. For users of the LINC platform, authentication is mostly transparent too: all they have to do is to provide a user name and password when logging into the LINC tools.

A related safety issue is the security of actor communication against eavesdropping and/or tampering with the data passed as part of task invocation messages. If security is an issue, any information passed between parties (including the task names themselves) could be encrypted by the proxy before sending it. The best way would be to handle encryption at the level of RMI itself, making it transparent to both software developers and users. The required keys could be exchanged as part of the authentication procedure.

9.4 LINC tool support

Whereas the case studies presented in part III were meant to show that it is possible to use the LINC language to describe a real-world system using LINC contracts, the goal of developing a prototype compiler for the language was to show that contracts can indeed be compiled into a form that allows for verification of the interaction in real time. Such a compiler is not only required to check the syntax of LINC contracts, but also to generate the classes and interfaces that make up an important part of the implementation of the actors involved in the contract. In particular, such a compiler will generate the required starters and registrars (section 9.2.1) for all parties, and the proxies and protocol verifier (section 9.2.2). So far, a pilot implementation has been built and tested; see section 9.4.1 for details.

As the proposed language is not only a design language but also a language that can be used to support auditing, debugging, and maintenance, many more tools are needed to make the language work in practical situations. Other useful tools consists of a contract browser (section 9.4.2) and an actor browser (section 9.4.3); such tools can be used to search the Logistic Contract Space for relevant contracts and logistic actors. The last tool that will be discussed in this section is the journal tree browser (section 9.4.5). Of those tools, only the last one has been implemented as a prototype. Practical use of the LINC platform will certainly suggest further useful tools.
9.4.1 The LinC contract compiler

The LINC contract compiler translates a valid LINC contract into a set of Java interfaces and classes; most of these were already mentioned in section 9.2.1. Apart from the classes and interfaces that form the output of the compilation process, there are some predefined packages that make up a LINC application. These packages are therefore part of the basic compiler distribution. The generated interfaces and classes can be used as the basis of a service implementation. The RemoteServiceProviderImplementation and RemoteServiceClientImplementation classes can be extended to include the actual process logic of both parties. Of course, contract parties can use any existing or newly developed code with the LINC platform; the only requirements are that each party is a subclass of LincRemoteActor and that it implements that party's RemoteServiceProvider or RemoteServiceClient interface.

Since the current compiler is only a prototype that is based upon a language version that has since evolved into the language described in this thesis, it is rather limited in many respects. At the time of writing, the most important restrictions are:

- only 2-party contracts can be compiled and verified,
- the monitored and timed interaction patterns are not implemented,
- subcontracted interaction patterns are not implemented
- task pre- and postconditions are not implemented,
- service objects, parameters, and indexes are not implemented,
- assertions and updates are not implemented.

Despite the fact that these important features are still lacking, the compiler has been successfully used to generate Java objects that can communicate over a network while the communication between them is being verified against the protocol and their interaction is logged in a journal tree.

9.4.2 The service browser

Being a hierarchical name space, the Logistic Contract Space can be represented by a tree-like structure that is browseable by a tool that is similar to many common browser tools. Using such a browser tool, users would be able select a particular service domain and view all service contracts defined in that domain. Filters could be used to restrict the displayed contracts on the basis of their name, domain, actor types, service object types, and possible (textual) attributes such as location, user rating, etc. Of course the browser would be able to retrieve the source text of the contract as well as (part of) the code of its implementation (the latter might be controlled by access rights).
Having identified a contract, the browsing party would be able to list all parties that are available as actors in the contract. As explained in section 9.1.4, the contract has to be concrete in order to be eligible for execution. This means that all contract parties must either be downloadable or able to being contacted. To visualize the status of contracts, the browser might use the 'red, yellow, green' metaphor: a contract's status would be green if all its actors are concrete (available); yellow if there is just one actor missing, and red if more than one actor is still abstract. When a contract's status is green, it can be executed directly, without a search for additional parties. When its status is yellow, a party of the right type could change the colour to green: this is especially interesting if the requesting party is of the right type (the yellow status could in fact be restricted to this case). If a contract's status is red no single party could change it to yellow.

The browsing process could be supported by a kind of search engine using information from stored contracts to match actor types and service object types specified by the user. In addition, it might be very helpful to include key words, key phrases, and short descriptions in the contracts to help users find contracts. The LINC language might need to be extended for this purpose.

### 9.4.3 The actor browser

Whereas the previous section mentioned the browsing of service contracts using a selected contract as a starting point to list logistic actors that are capable of playing a certain role in that contract, the opposite is also possible. Users would then list all known actors and be able to list all service contracts supported by a selected actor. Filters could again be used to restrict the set of contracts to those in a certain domain or those having certain attributes, and to select only contracts that use the selected actor in a certain role. E.g., one could search for all logistic actors that can act as a provider of a certain service in a certain area; this use is very similar to the way we use the traditional yellow pages directory. In our system though, providers could use exactly the same service to look for possible clients!

### 9.4.4 The contract broker

Given the structure of the Logistic Contract Space and the browsing tools just described, it is possible to design a brokerage service that can be used to locate the most suitable actor to provide or consume a certain service. Suitability might be determined by the actor's location, as well as it's availability, capacity, speed, and reputation gained from other clients. The broker could continuously search the net for parties that offer and/or require services, and instantiate them automatically if some specified set of criteria is met (e.g., parties would probably mark themselves as eligible for such a kind of brokerage). Brokers could be implemented by humans, by a piece of software only, or by a combination of both. Note that brokerage service itself might be described by a service contract in the Logistic Contract Space!
9.4.5 The journal tree browser

One essential tool in the LinC platform is a browser for journal trees. Such a browser can be used to visualize the interaction between logistic parties both when the contract is being executed and afterwards, after the contract has terminated. Section 9.2.4 already presented some simple examples of what such a browser might look like. A prototype browser was developed that is limited to displaying a graphical version of the journal tree. A more useful browser would allow users to selectively display nodes and to ‘open’ nodes to view additional information about the interaction pattern, such as time stamps and parameters passed as part of a task invocation message.

Experience has taught us that having such a structured dynamic view of the interaction is very helpful in understanding the interaction: the semantic structure of the journal tree directly reflects our understanding of the interaction (and hence of the software system), whereas interpreting a more traditional unstructured log requires a mental translation to be made from the sequential log to the structure of the problem space. A journal tree is therefore useful during the entire life-cycle of the application: development, design, regular operation, and maintenance. We therefore recommend the development of such a tool (the prototype might be used as a starting point) even more than the browsers we mentioned before. The figures 9.9 and 9.10 present some more snapshots of a journal tree that is in the process of being built. The snapshots have been taken from a real contract’s execution. The relevant part of the protocol is shown in figure 9.8: a number of parcels are announced and then either handled or canceled.

```
concurrent (theParcel : FreightUnit.Transport.Parcel)
multiple (50)
  ordered sequential
  client: announce (parcelDestination, clientParcel);
  select
    client : handle (clientParcel);
    client : cancel (clientParcel);
  end select;
end sequential;
end multiple;
```

Figure 9.8: Protocol fragment for figures 9.9 and 9.10

The first snapshot (figure 9.9) was taken shortly after the contract was started: the provider has been charged to handle a certain parcel set, and 4 parcels have already been announced. The multiple construct itself is not logged, so the various sequential constructs containing the announce/handle sequence are logged directly under the parent sequential construct. All sequential constructs are still open and active (see also section 5.13.9). The second snapshot (figure 9.10) was taken a few moments later. Now all inner sequentials have started, and a few of them already have a second task active. As the inner sequentials can only have two tasks, the latter sequentials have been marked closed (see below). The
first task of each of these sequentials has been marked as completed (see below), and a
placeholder marker for the journal value has been inserted (journal values have not yet
been implemented).

**Node attributes**

Each journal node is associated with one or more attributes. All nodes have an attribute
that shows if the construct is *active*, *inactive*, or *completed*, and *composite nodes* have an
additional attribute that shows if the construct is *open* or *closed*. The latter values are
defined as follows:

- an interaction pattern is marked *open* if it allows or needs more nested branches to
  be started.
- an interaction pattern is marked *closed* if no new nested branches can be started.

Note that this marker only applies to nested branches that may (or may not) be executed
*directly* within the enclosing construct; branches at deeper levels only affect the marker of
their own parent node\(^5\). Values for the former marker are defined as follows:

- an interaction pattern is marked *completed* if its execution has terminated (for simple
  nodes), or if all its possible (i.e., existing and future) nested branches are completed
  (for composite nodes). For composite nodes, this implies that the node is closed.

\(^5\)A newer version of the software also shows the sequence number of the current branch and the total
number of branches. This way, a distinction can be made between "allows more branches" and "needs more
branches".
Figure 9.10: Journal tree at $t = 2$

- an interaction pattern is marked *active* if it is currently executing (for simple nodes), or if it contains at least one nested branch that has started, but is not yet completed (for composite nodes).

- an interaction pattern is marked *inactive* if it does not contain nested branches that have started but are not yet completed. This rule implies that a construct whose existing branches are all completed, but which is still open to accept new branches, is marked inactive. This marker does not apply to simple nodes, which are active from start until completion.
Part V

Epilogue
The last part of this thesis presents an evaluation of the proposed logistic design method and the accompanying language. Starting from the requirements for the design as presented in chapter 4, chapter 10 will show to what extent these requirements have been met by our design. Chapter 11 will end part V by presenting our most important observations regarding the language and its use. This chapter will especially deal with how the case studies have influenced the language design: because the first case study was conducted at an early stage, its influence has been comparatively large. We will also show some areas in which the language needs further improvement; not because it cannot describe certain cases, but because the descriptions could be made more concise. Finally, this chapter will present a number of observations with regard to the style of writing service contracts.
Chapter 10

Evaluation of the design

This chapter will evaluate the LinC design and implementation. The evaluation will be based upon the requirements that were set out in sections 4.1 and 4.2 as well as the information presented in parts II, III, and IV of this thesis. In the following sections, we will first look at the language and its use to describe real logistic systems (section 10.1). Following that, we will have a quick look at the results that were obtained by means of our pilot implementation (section 10.2). The reader should keep in mind that most of our effort has gone into the former, and that the latter results are of a much more preliminary nature.

10.1 Using the LinC language

Whereas the previous sections defined a series of requirements to be met by the language and its implementation, we will now consider each requirement and see if and how it is met. Our validation will be empirical, i.e., we will report on the successes and failures of our use of the language to describe a real logistic system. The following discussion is structured along the list of application-specific requirements as presented in section 4.1:

10.1.1 Completeness of the description

As we saw before, our completeness criterion is defined in relation to the actor interfaces that are part of a contract. These interfaces specify the methods that have to be implemented by the various logistic actors, and are no less general than the interfaces that can be defined in many current programming languages. In contrast to most current languages though, the tasks that are specified in the actor signatures may be associated with pre- and postconditions. From a purely logical perspective, such conditions are enough to fully specify the required behaviour of each actor object. This behaviour is however described independently from the clients of an actor, and does not specify any requirements to potential client actors. Apart from this purely logical approach, which is often quite abstract, a service contract may include a specification of the clients as well as a usage protocol that
is less abstract and explicitly specifies the responsibilities of all parties involved in the contract as well as the interactions between different actor objects. As could be expected, the design of party signatures proceeds along the same line as the design of regular interfaces. Care should be taken in order not to over-specify the protocol, as the contract is not meant to restrict but to support the implementation. More about this under the next topic.

Two remarks should be made here: the actor interfaces (which are anonymous, i.e., not given a type name of their own) are defined in addition to the actor types that are specified as part of the contract description. As the language does not specify whether the anonymous interfaces are in fact implied by or even equivalent to the actor types, this specification may or may not be redundant. The implementation doesn’t care as long as all the specified tasks are indeed implemented by the actor object that is passed when the contract is instantiated. Another redundancy issue results from having both logical conditions and a usage protocol: conditions may be implied by or in conflict with the protocol, even though it might be difficult or impossible to see this from the contract. In our opinion, conditions that are implied by the usage protocol should not be repeated in the task signatures.

10.1.2 Adequacy of the description

The criterion of adequacy specifically applies to the usage protocol; it was introduced to replace the completeness criterion that was considered at an earlier stage of our research. In the context of programming languages, a language is called Turing-complete if it can be used to describe any computation that can be performed by a Turing machine. The usefulness of such a definition for our purposes is questionable, as the LINC language is not meant to be used to describe algorithms in the same way as algorithmic languages. More specifically, a LINC contract does not perform any visible computation on behalf of the contract parties. Instead, it describes the communication between actors that together execute some algorithm. The details of that algorithm and the internal state of those actors remain hidden, even from the protocol verifier.

It is the purpose of the LINC language to express any ordering or concurrency constraints that need to be put on the communication acts that take place between actors. The language must therefore be both expressive (in order to allow the designer to express the desired constraints) and descriptive (in order to assign a suitable structure to every valid sequence of communication acts). Given these requirements, the adequacy of the protocol refers to its effectiveness in analyzing and describing the semantic structure of interactions between the contract parties and to the relative ease or difficulty writing useful protocols. These issues will be addressed below:

\[^1\] Obviously, the protocol verifier performs some real computation: the results of this internal computation remain hidden in case the interactions are valid, or may become painfully visible to a party that is in error.

\[^2\] Imposing structure onto a ‘flat’ sequence of communication acts is in fact one of the major benefits of using protocols, comparable to the rules that govern human dialogue and attribute meaning to a sequence of individual utterances.
10.1. Using the LinC language

- *Expressiveness/decriptiveness:* In order to judge the effectiveness in describing communication, we have to draw a parallel between the algorithmic constructs of the language that is used to implement the actors and the communication that will result from executing those constructs. Only if such a parallel exists, there will be a reasonable correspondence between the state of the protocol and the internal state of the actors. In the absence of such a parallel, there is nothing to be learned from the protocol about the state of the computation. As the internal state of the actor objects is the result of executing the control structures of the implementation language, the interaction patterns should reflect those control structures. At the basic level, method calls to external objects are modeled by LinC task invocations (see section 5.13.1). The usual branching control structures are modeled by means of the optional and select constructs (sections 5.13.4 and 5.13.3, respectively). Iteration constructs can be modeled in LinC by means of the multiple construct (section 5.13.9). Groups of interaction constructs can be ordered sequentially or concurrently by means of the corresponding LinC constructs (sections 5.13.7 and 5.13.8). Finally, the language has constructs to describe an alternative path to be followed after an exception has occurred (section 5.13.5) and to specify timing constraints (section 5.13.6). Every common algorithmic construct therefore has a counterpart in the communication protocol. In principle, this protocol can therefore accurately reflect the state of the computation. One aspect that is shared by all LinC constructs is that they are intentionally less specific than the corresponding algorithmic language constructs. In particular, LinC does not allow the designer to express the exact conditions and choices involved. The reason is of course that LinC is a specification language rather than an implementation language; the LinC designer should use the language to provide a logical context for the interactions; not to force the actor implementors to re-implement a specific algorithm.

- *Protocol ambiguity:* There is an inherent problem with protocols that will be mentioned many times throughout the language definition in chapter 5: protocol ambiguity. In the normal course of executing a contract, every protocol state change is uniquely determined by both the state that is current and the task invocation message that occurs while being in that state. Protocol ambiguity happens when a certain task invocation can be executed in more than one protocol construct given the current protocol state, so executing the task would cause a transition to an undefined state (or to a transition to a set of states). If this can happen, the protocol is said to be ambiguous. After a transition to such an indeterminate state, the protocol verifier wouldn't know what to do with subsequent task invocations. Even more importantly: the receiving party might not know what to do with it! Even though technical solutions can be designed that allow such ambiguities to be eventually resolved, this is a good reason to disallow any protocol ambiguity\(^3\). Obviously it would be nice if the LinC compiler would generate a warning if a protocol were potentially ambiguous. Note that we say 'potentially': it is quite possible to have a protocol that runs into

\(^3\)It is for similar reasons that most natural language users are less than happy with an utterance that can only be understood after one or more successive sentences.
ambiguity for one series of task invocation messages, while being perfectly unambiguous for another series. Total safety would however force us to disallow any possible ambiguity, and might become too restrictive. The current system therefore uses a run-time check to see if a particular task invocation can lead to an indeterminate state. If so, an exception is thrown.

- **Use as a design language:** As the reader of part III will notice, designing service contracts can be a tedious task. While some protocols are simple, others may seem overly complex, especially when exception handling comes into play. This is not a problem of the language, though, but actually a virtue: it makes logistic complexity visible, and hence manageable. The designer is forced to think beforehand about how to handle the most common exceptional cases; once these problems are solved, the logistic design can be implemented smoothly. Compare this to an approach where the logistic design only covers the regular case: in such a system exceptions might be handled as an implementational afterthought, resulting in a system that is very hard to handle and maintain. In conclusion, the practice of designing the service contracts of part III has taught us that while designing contracts may seem hard at the beginning, experience begins to pay off quickly while leading to designs that are quite robust in the face of common disturbances.

As a final note, we should mention the existence of a certain field of tension between logistic designers and software engineers. Whereas the former group would like to limit the design to just the core of the logistic system, the latter group is convinced of the necessity to specify certain details that do not seem immediately relevant to the system's logistic functions. With regard to this dispute, we tend to conform to the latter view ourselves, but accept the fact that there are arguments in favour of the former.

### 10.1.3 Contract verification

With contract verification we refer to checking actual communication between logistic actors against the usage protocol. Interactions that violate the contract are not allowed, and the offending party will be informed. In order to allow for this kind of checking, it must be possible to match basic interactions (i.e., task invocations) to the protocol part of the logistic contract. If a given task invocation does not match any corresponding interaction pattern in the protocol, it is marked invalid.

Even though the current pilot-implementation does not yet cover all possible interaction patterns, it shows that such verification is possible, at least in principle. There have been a small number of cases, though, where we were forced to adapt the language design in order to allow a construct to be verified. The *multiple* construct (section 5.13.9) is a good example: the multiple index was only added after it was discovered that without it, multiple constructs are ambiguous by definition. Furthermore, after its introduction it was not immediately realized that service variables needed to be indexed by the multiple index. While this particular case might be considered a gross oversight in the original design followed by
the premature addition of a new language feature to fix it, most language/implementation conflicts appear to be much more subtle. E.g., the need to pass the multiple index to nested subcontracted interaction patterns (see sections 5.10 and 5.13.9) has not yet been resolved: probably we can do without service indexes. What we can learn from these examples is that problems like these are bound to occur if we try to implement a language before its semantics are formalized. As always, the reason we didn’t proceed that way was just a matter of priorities.

10.1.4 Ability to handle exceptional situations

This criterion was mentioned separately because the handling of exceptions forms a considerable part of many contracts. For the purpose of evaluating the different language constructs, most aspects of exception handling fall under the header of adequacy. We’ll therefore now limit ourselves to some aspect that is specific to exception handling, viz. exception recovery. Originally, there was only one way to continue after an exception had been thrown: adhering to the protocol structure that was specified as a handler. This meant that to really recover from the problem, the exception handler itself had to retry the problematic interaction, with the risk of running into the same problem of course. And so on. This prompted the design of a more elaborate exception handling mechanism, including the possibility to invoke special handler tasks from the protocol verifier itself! This is the only case where tasks invocation messages do not originate from one of the contract parties; see section 5.13.5 for further details.

10.1.5 Ability to escape from the contract

It would be a huge mistake to think a contract can ever be complete and that every possible exception can be foreseen and handled under the control of a precooked protocol: the range of problems that can occur exceeds all human imagination (except maybe that of Murphy’s [Off78]). In our logistic world this means that one has to be able to escape from the control of the contract and fall back to more traditional means of communication and rely on humans who can improvise to achieve the best results given the unfortunate situation they are confronted with. The mechanism that is available to the logistic designer to allow for this ‘escape’ is again an exception. Only this time, the exception is not a specific one, but one that is generic: ProcessError or a subclass of it. If handled, the specified handler cannot do anything but call upon one or more actors to solve the problem inside a special exception handling task. If not handled, this generic exception just terminates the contract, in which case the platform software that was used to instantiate the contract (see section 9.2.1) can try something similar or just file a legal claim.
10.1.6 Structured logging

Apart from a number of toy contracts, we have tested and demonstrated the SinglePlatformMove and MultiplePlatformMove contracts of sections 7.4.4 and 7.4.3, respectively. While this test didn’t lead to any particular surprises, one result is worth mentioning: the possibility to see a journal tree being gradually built (the results of these tests were very similar to what is shown in the figures 9.9 and 9.10). This journal viewing facility appears to be a very useful aspect of the platform and its associated tools. This is because the designer of a system can now see the interactions in their proper context instead of having to do a mental ‘transposition’ to put them into context herself. This leads to significantly better understanding of the system and is an extremely valuable aid in debugging.

10.2 Using the LinC implementation

In this section, we will present the implementation related evaluation criteria. As we have said before, the current implementation is just a pilot-system, and therefore not all features of the language are implemented (this holds in particular for those features that were changed or introduced during the course of the case studies). The compiler and contract browser have however been used to test some of the Cross Docking contracts presented in chapter 7. The first two criteria that we’ll mention here have to do with contract rehearsal and with the way in which potential partners in a contract can locate each other and initiate their cooperation. While we used to treat these aspects as language criteria, the implementation issues are actually more important than the language issues.

10.2.1 Contract rehearsal

Contract rehearsal requires that the implementation is able to generate dummy actors, given the information contained in the service contract. Although generating simple dummy actors is straightforward, the real question is if we can generate dummy actors that fully exercise the interaction protocol. At this time, we have not researched this topic in any detail, so this question still remains.

10.2.2 Contacting partners

Although the mechanism for finding potential contract partners and then contracting them is one of the major components of the LinC architecture, this topic played a relatively minor role in our research, mostly because it seems more of an implementation/design issue than a research issue. It is therefore impossible to base our conclusions on the current pilot implementation. While this pilot system uses regular Jini lookup to locate a party on the basis of its signature (as generated by the LinC compiler), any realistic implementation has to do a better job. In particular, we have to resolve some issues around the exact
10.3 Concluding remarks

contents of a party’s signature, which could include the contract name, the actor type, the implemented interface(s), the actor's geographic locations, some indication of the actor's availability, and even some kind of user rating!

10.2.3 Standardization

The current implementation was totally based upon open standards and free software: Sun Java and Jini. While future versions might need additional software systems to implement discovery (see section 8.6.1), it is likely that both free and commercial ('enterprise') versions will be readily available. In addition to these free software platforms, it should be fairly easy to support e.g., Microsoft's .NET platform, either in a single language environment or in a mixed language environment (using SOAP).

10.2.4 Security

We can be short about this one: the current definition of the LiNC language and system has no special provisions to enforce security, and the current implementation of the LiNC compiler does not do anything to authenticate parties and protect transmitted data. Given the current Java based implementation, implementing security should be straightforward: as from version 1.4, the Java 2 SDK provides the Java Authentication and Authorization Service (JAAS, [JAA]). Also, data being sent in remote method invocations can be protected by means of encryption techniques; encryption can be built into the LiNC level software, be obtained by using RMI over secure sockets, or be implemented by means of the Extensible Remote Invocation system, which is an extension of regular RMI that is available in the form of the net.jini.jeri package. Security will therefore be an important concern in upcoming versions of both the language and the compiler.

10.3 Concluding remarks

The core of our work has been related to the design and use of protocols to describe the interaction between logistic actors (or software objects in general). The protocols that we propose constitute a departure from state-transition based descriptions like used in some other approaches (see section 1.6). We think that the grammar based approach that underlies our protocols offers a more transparent and intuitive way to describe object interaction.

In addition, our research has focused on the use of contracts as service signatures to be used by parties looking for partners in a virtual enterprise. We are convinced that the use of contracts to locate actors is to be preferred to the common type-based or 'attribute-value' pairs approach to service discovery. In our opinion, the two approaches can (and should) be merged: the service contract is to become an attribute value; additional attribute-value
pairs can then be used to specify the details of the actor that is publishing itself.

Regarding the implementation, we conclude that certain implementation choices have been made during the course of this work. Not all of these choices are meant to be definitive; in particular, the communication mechanism used in future implementations will likely be extended to include message-based interaction as well. Alternatively, the current method-based mechanism might be extended to include asynchronous method calls (meaning that the method invocation and return might be queued). This does not affect the grammar-based protocol description that we consider the core of our work.

Similarly, the choice for a language-centric approach should not be taken as a rejection of platform-independent approaches (using e.g. XML-based communication and taxonomies to define a common vocabulary). Most of our work applies to either approach, and it might be possible at some point to allow a mixture of Java and non-Java actors to execute a service contract together. Also, the choice for a particular discovery mechanism is not central to our approach. It is almost certain that future systems will use a combination of different discovery techniques.

In conclusion, we think that the core ideas of our approach will stay intact, even when a number of particular design and implementation mechanisms will be replaced at some later stage. In fact, this is the only way to survive the effects of time: a system can either evolve or die, there is no in-between.

These conclusions lead us naturally to the final question: what about ERP and the systems that are in use right now? As others have already argued (see e.g., [Kay03]), loose coupling is bound to replace tightly coupled systems. The effort and cost of integration of ever more subsystems into the enterprise system will become prohibitive and the expanding system tends to become more inflexible over time. Also, in the case of supply chain systems, there might be political and security reasons that prevent the various provider's systems from being integrated. This is not to say that ERP systems will be obsolete any time soon. The limitations clearly suggest a possible way out, namely the loose coupling of more limited-scale (existing) ERP-based systems. There is no reason not to encapsulate an ERP system inside a logistic actor, allowing it to take part in a logistic operation defined by a LINC contract. The actor would decide how much of the ERP system would be visible to the outside world. Used this way, ERP systems may continue to play a role in future logistic systems.
Chapter 11

Observations: the language and its use

Even though the first case study was of a rather limited scale, it has suggested a fair number of additions and changes to the language as it was at the time the study was started. The second case study, being significantly more complex, confirmed the usefulness of the language changes that had already been effectuated. Nevertheless, the larger scale of the second case study has caused some other problems to surface, and has inspired some further changes. To give the reader more detailed insight into the evolution process that lead to the language as it is now, we will present the most interesting additions and changes in section 11.1. All these additions and changes have already been incorporated in the language as it is defined in part II; also, the service contracts in the current part have all been adapted to reflect the current version of the language. Some issues that have not yet been resolved or that need more work in order to be defined in a way that is consistent with the rest of the language will be presented in section 11.2. Finally, section 11.3 will present our most important conclusions with regard to using the language to define logistic systems.

11.1 Additions and changes motivated by the case studies

Work on the first case study was started soon after the initial version of the language (still called LOGOS at that time) was completed. The goal of this early start was to create an extended example, and to get feedback at an early stage; indeed a number of difficulties and shortcomings were identified. This section will describe the most important additions and changes that have already made it into the language, together with the problems that have inspired them.

11.1.1 The number of service objects

Initially, only one service object could be passed to a service contract: the idea behind this restriction was that every contract is concerned with a specific ‘thing’ that is the object

\footnote{Note, however, that most of the changes have not yet been implemented in the LINC compiler.}
of the service, i.e., the thing that is ‘handled’ by the service. The service object was to reflect the state of that ‘thing’ and make it available to external (and internal) observers. In practice, it appeared that there are often several kinds of things that are handled by a contract, so the restriction to allow only one proved too much a limitation. Because the number of service objects is totally independent of the rest of a service contract, the syntax could easily be changed to allow for more than one service object.

11.1.2 Service contract parameters

It was soon recognized that many contracts might benefit from a way to pass operational data (such as numbers and times to be used as constraints, actors to be used as subcontractors of services, etc.) upon contract instantiation. Even though it would be technically possible to use service objects to pass such data to a contract, this was considered kind of abuse: operational constraints do not reflect any execution state and should be prevented from being updated during execution of the contract. For these reasons, a parameter section was added to the language instead. The main difference between service objects and service parameters is that the former can be used in updates (see section 5.13.11) while the latter cannot: service parameters are are strictly read-only once they are bound.

11.1.3 Default values for service objects/parameters

The practice of writing real contracts has shown that sometimes the service objects and/or parameters are not yet known at the time of contract instantiation. Fortunately, the logic variable semantics of service objects and parameters makes it possible to leave them unbound until a value is assigned (e.g., during contract execution). An unbound service object or parameter can be assigned a value by referring to that object or parameter in a task invocation message, usually during the initial stages of the execution of the contract. Of course, making the initialization of service objects and parameters optional introduces the risk that the service object or parameter is still unbound when a value is needed. This happens, e.g., when it is used to constrain a multiple construct. Since it is not always possible for the compiler to predict if the service object or parameter will be initialized in time (a warning might be the best it can do), a runtime exception is required to capture any erroneous situations that might have slipped through. As this is not a situation that can be fixed, this exception will probably fall into the same category as ProcessError and TimeoutError, meaning that the contract will be aborted.

Since delaying the binding of service objects and parameters is an important design decision, it was decided that contracts should make a clear distinction between bound and (possibly) unbound service objects or parameters. Instead of introducing a special marker, it was decided to use another new mechanism: default values for service objects and parameters. If contract writers do not specify a default value for a service object or parameter, a value must be specified when the contract is instantiated. If they do specify a default value for a service object or parameter, a value need not be specified at contract instantiation. If
a value is still specified at contract instantiation, the latter value is said to override the default value (which is then completely ignored). In all these cases, the service object will be bound when contract execution commences. A special initialization value, null, can be used to specify that the service object or parameter is intentionally left unbound\(^2\).

11.1.4 Delegation

One issue that played a role during an early stage in the design of the LINC language is delegation. Early publications [ELL99, ELL00, Eve00] referred to delegation contracts as a special kind of contracts that were concerned with the coordination of activities taking place in a number of separate service contracts, especially in cases where activities in one service contract occur as the result of executing a certain task under control of another contract. Because such a relation typically occurs when a party delegates all or part of its job to one or more third parties, such contracts were called delegation contracts.

A major problem (to the author of this thesis, at least), was that the meaning and use of such delegation contracts was not entirely clear. From the beginning it was obvious that contracts with more than 2 parties could be designed as a generalization of regular 2-party service contracts: section 5.16 describes the result. The proposed multi-party contracts indeed allow subcontracting behaviour of contract parties to be controlled from outside (e.g., by the client). However, multi-party contracts do not make the idea of delegation explicit in their design. Furthermore, we tried to use the 2-party contracts presented above\(^3\) as a starting point for the design of a single multi-party contract that incorporated the entire design. As could be expected, the resulting multi-party became quite intricate and needed heavy indentation. Even worse, the language forced us to replicate parts of the protocol in different branches of enclosing select constructs. The net effect was that the resulting multi-party contract obscured more issues than it clarified; obviously, some other solution was needed.

The new solution consisted of the process block, a construct that can be used to include some of the internals of a contract party's operation into a regular service contract. In practice, process blocks will mainly be used to specify that services are to be subcontracted from a third party, using one or more subcontracted interaction patterns inside the block. As a process block can contain any interaction pattern, it is also possible to use task invocation messages inside a process block. Since the facility is meant to describe delegation, the sender of a task invocation message and the client of a subcontracted interaction pattern must always be the party that 'executes' the process block. The receiver of a task invocation message, on the other hand, may be any 3rd party specified as an actor in the (multi-party) contract; the provider of a subcontracted party may be a multi-party actor, or the (actor-typed) value of a bound service parameter or bound service variable\(^4\).

\(^2\)Currently, null happens to be the only default value that can be specified; the language must be extended to allow for more complex initialization expressions.

\(^3\)Note that the language did not yet contain assertions, updates, and process blocks at that time.

\(^4\)These actor restrictions are purely meant to confine the use of process blocks to describe delegation;
For now, process blocks are just treated as comments, i.e., they are skipped by the compiler, and hence completely ignored by the protocol verifier. Future implementations might decide to do something useful with them, e.g., a process block could be treated as an anonymous in-line task whose internals are checked by the protocol verifier. In particular, their beginning and ending should be monitored (see also the next section). Having taken the step to include in-line tasks, we might as well decide to include named in-line tasks as well.

11.1.5 Nested concurrent constructs

As an immediate consequence of the introduction of process blocks into the language, we needed a way to show that a process blocks provides the implementation of a specific task. Enclosing a task invocation message and a subsequent process block in an ordered sequential or a ordered concurrent will not work, especially if the task has a post-condition that must hold upon termination of the task, and hence the process block. A new construct, nested concurrent, was added to the language for cases like this. This construct specifies that a construct that follows another construct must start after the construct that precedes it and that it must end before that construct. This also allows the process block to include assertions that directly match the pre- and postconditions of the preceding task.

11.1.6 Process error

If process blocks are like unnamed tasks, one would expect that they, like tasks, might end by throwing an exception. There is no obvious place where to specify such exception(s), and it was therefore decided to treat process blocks like regular protocol constructs, even though they actually form a 'window' into the party's implementation. As a result, process blocks are defined to propagate any un-handled exceptions coming from constructs inside the process block (see also section 5.13.12). These exceptions can then be handled by a regular exception handler outside the process block5.

Since process blocks are only a partial specification of what goes on inside the executing party, exceptions might also be thrown from activities that are not described in the process block. Since we would not know the specific exceptions in such cases, but might still be required to handle them, a special exception was introduced: ProcessError. Any exception not originating from a specified task invocation message inside a process block will be translated to this new exception. The ProcessError exception can be handled like any other exception; unlike regular exceptions though, it does not have to be handled, so even a subcontracted service might end by throwing it. At a later stage, it was decided that regular tasks may also end by throwing this exception. It is not required (or even possible) to specify the ProcessError exception in a task signature. Note that the throwing of a

---

5An alternative would have been to forbid any exceptions coming from a process block, and to require a handler inside the block, or to transform all exceptions into a ProcessError.
ProcessError will be considered contract breach: it is up to the parties' lawyers to decide who is right and what to do next.

11.1.7 Subcontracting inside multiple constructs

The initial version of the language did not allow for subcontracted interaction patterns inside multiple constructs. To remain consistent with the rest of the language, a separate instance of the sub-contract would need to be created for each branch of the multiple construct. Furthermore, the protocol of the sub-contract would need to replace the subcontracted interaction pattern for the purpose of contract verification. While this may look simple, it implies that all task invocation messages exchanged between the actors as part of the sub-contract must not only be interpreted within that sub-contract itself, but also at the level of the parent contract(s). It also implies that all task invocation messages (and hence the task signatures) in the sub-contract have to define an index argument corresponding to the multiple index that is defined by the parent contract! This was considered unacceptable at the time.

In practice, disallowing subcontracting inside multiple constructs turned out to be too severe a restriction: even the simple case study of chapter 6 already needed a way to instantiate sub-contracts inside a multiple of the main contract. The adopted solution was to add a service indexes section to the language, thereby making it possible to pass the multiple indexes of the main contract to subcontracted interaction patterns just like they are passed to simple task invocation messages. If a service contract has such an indexes section, all its task signatures have to include the corresponding index argument(s), and all task invocation messages have to refer to these service indexes.

Although the proposed solution works (on paper, at least), there are good reasons to criticize the need to pass multiple indexes to nested contracts: from the viewpoint of the sub-contract, these indexes seem irrelevant and only present for reasons of implementation; if it were possible to ignore the outer contract while checking communication taking place as part of the inner contract, there would be no need to pass multiple indexes at all. Further research is necessary to determine if this requirement can be dropped.

11.1.8 Exception handling

Even though the try construct has been part of the language from the very beginning, its use was initially more limited than it is now. Originally, exception handlers could only contain a regular protocol construct, thereby providing a way to specify an alternative continuation of the protocol. The handler could decide to do nothing or to allow the parties to communicate in order to fix the problem. In many cases, this meant that the handler not only had to fix the problem, but also had to re-do the protocol construct that was at the origin of that problem! This made exception handlers difficult to design. A further complication was the fact that the state of execution at the time of the exception might
not be known, so it might actually be impossible to fully recover from the exception. We then realized that the difficulties were related to what we since have since called formal vs. informal exceptions: whereas formal exceptions allow meaningful continuation of the service, informal exceptions do not allow for that, and are just meant to escape from the contract and rely on other means of communication between the contract parties. That is why they usually consist of a broadcast to all parties, followed by a premature ending of the contract.

The various ways to terminate an exception handler (retry, resume, restart, and reject, see section 5.13.5) were also introduced to improve the situation and to allow contract writers to state their intentions clearly. Practice has shown that these constructs helped a lot to shorten the contracts and to avoid unnecessary complexity. As these termination 'commands' have not yet been implemented, nothing is known about the burden they place on the compiler and the runtime system; since most of them imply some kind of roll-back of the protocol, the contract parties must be prepared to handle roll-backs, a task that might be quite difficult to achieve.

11.1.9 Timing requirements

When the first case study was started, there was nothing in the language to specify any timing requirements. Tasks or sub-contracts could take forever, and the time interval between completion of one task or sub-contract and the start of the next could become indefinitely long. For the parties involved, it would be difficult or impossible to decide if the party they are waiting for is just a little busy, or if it is gone forever. For the same reasons it would be impossible for clients to decide if a provider task takes too long to complete. For these reasons, another special exception was added to the language: TimeoutError. If a contract party has waited too long before it receives a new task invocation message, it is allowed to throw a TimeoutError. Again, it is neither required nor possible to specify this exception in the task signature. It is up to the party itself to decide if it has waited long enough. Note that the throwing of a TimeoutError will be considered contract breach; it is up to the parties' lawyers to decide who is right and what to do next.

Because a TimeoutError can hardly be considered an elaborate means to handle timing requirements, it was decided at a later stage to add the timed interaction pattern to the language (see section 5.13.6). This interaction pattern provides a little more flexibility, although still less than we would like. In particular, the timed interaction pattern is passive, like all other interaction patterns. It therefore cannot issue an advance warning to a party that is taking too long, and violations of the timing constraints can not be detected before it is too late. Even then, it can not do anything better than throwing an exception; more research is necessary to expand its use.

Another option that was considered for a while was to introduce timing constraints in the form of timing pre- and postconditions on tasks and sub-contracts, and possibly on other interaction patterns as well. To make such conditions usable, it would be necessary to
introduce something like time variables into the language. This option was rejected in favor of the timed construct, which is orthogonal to the other language mechanisms.

11.1.10 Assertions/updates

In the earlier stages of the design of the LINC language, service objects could be passed to a contract, but not used in any way other than passing them to task invocation messages. As it is prohibited to update arguments from within a task, service objects could not be changed at all, and therefore could not be used to represent the state of execution of the contract.

One way to make service objects more useful would have been to allow tasks to update arguments that refer to service objects. This option was rejected for many good reasons: making the semantics of parameter passing dependent on the kind of object passed is confusing, and the implementor of a task would need to be aware of the difference between task arguments that can be updated and arguments that can not be updated. Even worse, this would depend on the calling context. Finally, this way of updating service objects would conflict with logic-variable semantics. A different approach was therefore chosen: no party can change a value; only the protocol manager can, by means of a new update construct. This new construct has many advantages: service objects are read-only for the actors, as they should be; service objects are updated from within the protocol itself, so the time at which parties may observe their value being changed is well-defined; the effect of the contract on the service objects is made explicit.

11.1.11 Premature contract termination

A final point that must be mentioned is that parties are not free to step out of a contract while it is in execution. The reason is obvious: after having accepted a contract, parties have committed themselves to perform the tasks that are specified by the contract. If they withdraw, legal action might follow. Currently there is nothing in the contract to specify what the consequences might be if parties do not live up to their commitment; actually, no decision has yet been made if this should be part of the contract at all, or if it should be part of the ‘environment’. Whatever way the sanctions are specified and applied, the protocol verifier should have a means to determine if a party withdraws prematurely.

A possible way to make sure that the entire protocol is executed is to add explicit charge* and discharge* phases to the protocol, as was the case in the initial design presented by Evers in [ELL99]. Parties would not be allowed to interact until the charge phase has completed, and they are not allowed to back out before the discharge phase is over. Of course they can still do so (by just aborting themselves), but if they do they will have to face the (legal or other) consequences. Another way is to introduce implicit charge and discharge phases at the beginning and end of the protocol. The LINC runtime system would have to wait until all contract parties have sent a charge () message, and then broadcast
a kind of open message in return. At the end of the contract, each party would have to send a discharge message; after all parties have done this, the system will respond with a close message. Any party that withdraws from the contract before it got a close message is in violation of the contract. The latter functionality is implemented in the current runtime system: the basic actor (see section 9.3) provides a default implementation for charge and discharge, which can be overridden if necessary.

Even though the proposed scheme can handle premature termination of the protocol, it cannot distinguish between accidental interruption of the communication flow and parties that deliberately terminate the interaction. More work is needed to design a way to resume a contract in the case of accidental and/or temporary interruptions.

11.2 Open issues regarding the LinC language

Whereas the preceding section listed the most important issues that have influenced the language design and that have already been incorporated into the current LinC language, there are also some issues that are still unresolved. The most important ones will be discussed shortly in the following.

11.2.1 Multiple indexes and exit constraints

The current LinC language requires that every multiple construct that contains something else than a single task invocation message has a multiple index defined for it (see section 5.13.9). In many cases there will be a task argument that is an obvious candidate for use as the index argument (see section 5.13.9); in other cases, there is no such candidate, in which case it is necessary to ‘invent’ an argument especially for this purpose. This is especially annoying when the multiple construct is inside a process block or when it contains a process block. In such cases we would like to avoid the need to come up with an index that is actually never used (section 7.4.1 provides a good example). The same section also made it clear that it is indeed desirable to be able to exit a multiple implicitly, i.e., by sending a task invocation message that can only be interpreted outside the multiple.

11.2.2 Pre- and postconditions on service objects

As shown in section 7.4.2, it would also be useful to have pre- and postconditions on the service objects. Although the meaning of such pre- and postconditions is intuitively clear, there are still many questions to be answered before they can become part of the language: when will they be checked? What should be the effect if they fail? How do they interact with default initializations? For now, only the first question has a clear answer: they may be attached to the contract’s charge and discharge (see section 11.1.11). The answers to the other questions are less clear, so until these matters are resolved the best
advice may be to include comments in a contract containing pre- and postconditions.

11.2.3 Pre- and postconditions on sub-contracts

The lack of pre- and postconditions on service contracts as a while makes it often quite difficult to understand the effects of executing a service contract, especially when the specified service is being subcontracted. This absence of postconditions means that one does not know anything about the state of execution that exists after the sub-contract has terminated. All we know is that any named exceptions thrown inside the sub-contract have been handled; however, there is no guarantee that these exceptions are handled in a way that ensures the same (unspecified) postconditions as normal termination would. In fact, letting a contract throw an un-handled ProcessError exception might even be safer!

As a partial remedy, one may of course include assertions at the beginning and end of each subcontract. Such assertions do not offer a reliable solution to our problem though, mainly because violations of the final assertions would not be caught inside the sub-contract itself (see 5.13.10). This means that the sub-contract has already terminated when the problem is finally detected! Another problem with the inclusion of pre- and postconditions on (sub-)contracts is that, unlike similar conditions on tasks, it is not clear who to blame if they failed, or what to do in response. More research is needed to design a solution for these problems.

Discussions on the topic of pre- and postconditions on sub-contracts have not yet been resolved: one position is that sub-contracts are usually instantiated inside process blocks and that they are therefore part of a party’s process logic, which cannot and should not be fully described. While this argument holds some ground, we still feel that service contracts as a whole should have explicit pre- and postconditions, if only because they can also be used outside process blocks!

11.2.4 Transactions

In the current system, interactions are not guaranteed to terminate in a well-defined state: there might be circumstances, such as a loss of the connection between systems that are housing the various actors, that cause the system to run into a consistency problem. Transactions may offer a solution to a problem like this, since they would force an interaction to end in a valid state: either the state as it would be reached after the interaction was completed, or the state that existed before the interaction has started. The current system does not treat interactions as transactions, though, so an interaction might stop in the middle leaving the system in an unknown state. Note that transactions can be used at different levels: not only can each task invocation message be executed as a transaction, it is also possible to execute the higher level protocol constructs as transactions, rolling back the protocol to the initial state of a construct if something should happen in the middle.

Instead of introducing new variants of the existing protocol constructs, it would be better to
extend the language with a single new transaction-like construct that can have any protocol construct nested within it. That nested construct would then be executed as a transaction\(^6\). Staying in line with the current LiNC style, such a construct could be designed as a **guarded** ... **end guarded** block. This construct is not yet included in the current language, though.

One complication needs still be mentioned: as stated by e.g. [Kay03], conventional ACID-style\(^7\) transactions are not very suited to loosely coupled system as they may require resources to be locked inside the actors. While this is not a problem for a synchronous interaction style, it will be for asynchronous interaction since by definition the time between related interactions is not predictable. Locking resources for an unknown period is clearly unacceptable.

### 11.2.5 Actor restrictions on sub-contracts

The current context dependent restrictions on the actors of subcontracted interaction patterns (see section 5.13.2) might me too harsh and actually unnecessary. Even though subcontracted interaction patterns are often used to describe delegation, this is not the only context in which they are useful. Examples can be found in the sections 7.4.18 and 7.4.19. Apart from lifting all restrictions on the actors of sub-contracts, we might decide to slightly change the syntax of subcontracted interaction patterns in a way that shows their role, e.g., apart from the keyword **new**, extra keywords like **delegate** might be used to indicate sub-contract roles.

### 11.2.6 Separation of instantiation and execution

The contracts presented in this case study contained several examples of sub-contracts that are referred to from more than one context. There is currently no way to indicate that the various occurrences all denote the same instantiated contract, or to make a distinction between the instantiation of a new contract and the reference to a contract that is or will be instantiated elsewhere. In addition, the referring context might lack the information that is needed to fully specify the subcontract. The **iptBrokerage** is a good example of the problem: this contract refers to services that will be instantiated later. In fact, these references are meant to restrict the actual services the IPT can be asked to provide to those services that are explicitly specified in the brokerage contract. In other contexts (e.g., figures 7.12 and 7.18) the multiple references also serve to show that the IPT is rented to perform a specific service for the client.

To overcome the problems caused by multiple references, the language could be extended with parameters and variables taking sub-contracts (actually references to sub-contracts) as their values. This extension would need to allow sub-contract references to be passed around

---

\(^6\)Considering the problems with subcontracting, a sub-contract invocation is an obvious candidate to being executed as a transaction. A failing assertion could then cause the transaction to abort.

\(^7\)The ACID acronym stands for Atomicity, Consistency, Isolation, and Durability; see also [HR83].
between contracts. Like other parameters and variables, sub-contract parameters and values should be typed, allowing only specific sub-contracts as their values. Subcontract types could be simple (allowing only a specific service) or complex (using signature matching to check if a certain sub-contract is a proper value for the parameter or variable). In addition, instantiation and reference should be distinguished by means of different keywords, e.g. **new** for an instantiation vs. **use** for a reference. The latter change might even prove a useful extension per se: the **use** context might be used to relax the syntax rules for sub-contract instantiation and allow for missing service objects, parameters, and indexes.

### 11.2.7 Concurrent use of shared resources

The current language simply ignores the problem of concurrent access to shared resources. Section 7.4.6 provides a good example: the central **APS** is a provider that provides many instances of the **palletStoring** and **palletRetrieving** simultaneously. In order to function properly, the **APS** must ensure that at least one stack position is free when a pallet needs to be stored, and that at least one stack position is non-empty when a pallet needs to be retrieved. However, the **LINC** description is not able to capture the effects of several clients contending for the same resource, and the inclusion of assertions in the protocol will not have the desired effect: the test that would be performed to check that a resource is free is separated from the claiming of that resource, with the result that something could happen in between. This is what has been called the **concurrent precondition paradox** [Mey97, page 994]. In our case it means that some other client may have claimed the wanted resource, even after our precondition has been met. Just disallowing the provider to serve more than one instance of the contract won't work since there may be different service contracts that also need the same resources.

The problem described above is inherent to the design of concurrent objects. The usual solution to concurrency problems is to introduce critical regions that force the client to obtain a lock before the region can be entered. Usually, the object to be accessed is locked by means of special syntactic constructs (e.g. **synchronized** [Gra97]). Explicit **semaphores** are another option (see [Hol00]); an extension of the regular semaphore is adopted in **TrAcEs** [Lin03]. Meyer [Mey97] argues that pre- and postconditions are still the best way to enforce concurrency constraints. However, to avoid the concurrent precondition paradox, the semantics of pre- and postconditions have to be changed. To use his proposal in the **LINC** language, it would not be sufficient to change the semantics of pre- and postconditions: assertions and updates should be adapted as well.

Another way to solve the problem is to lock the resources that are needed by a contract for the duration of that contract's execution, and to prevent the instantiation of service contracts that need the same resources. An obvious resource to be locked is the service object (see also section 6.2.4) there might be other resources that also need to be locked. This mechanism is quite restrictive though, and could have a serious consequences for the provider's performance.
As a final note, we should mention that according to Evers [Eve04], concurrency constraints should be considered part of the actors’ process logic, and should not be integrated into the contractual specification. At this stage, we tend to a different view; further research is needed to decide on this issue.

11.3 Using the LinC language

In this section we will present a number of observations that are related to using the language to specify real logistic systems. One of the issues, that also arises in ‘regular’ design-by-contract, is related to the choice between preconditions and runtime exceptions; related to this issue is the use of assertions in process blocks. Finally, we will look a little deeper into the way in which errors can be prevented and/or handled. Note that this section is not meant to evaluate the LinC language as a means of designing logistic systems; that kind of evaluation is the subject of chapter 4.

Use of assertions in process blocks

Since process blocks are often used to provide additional detail on the implementation of the preceding task invocation, it is good practice to repeat the task’s pre- and postconditions as assertions at the beginning and the end of the process block whenever possible\(^8\). This is especially true if the body of the process block contains subcontracted interaction patterns, which cannot have pre- and postconditions of their own. By including assertions, the requirements on the embedded sub-contracts can at least be made explicit, even if they are not checked.

Use of assertions around subcontracted services

As we already said in section 11.2.3, one of the current language ‘problems’ appears to be the lack of pre- and postconditions on sub-contracts. Again it is good practice to include assertions instead (both at the beginning and at the end of each contract), so at least the intentions are clear. Doing this also stresses the fact that the ‘postcondition’ assertions must hold even if an exception has changed the protocol flow inside the sub-contract. Unfortunately, a failing ‘postcondition’ assertion will not be caught inside the subcontract, if at all.

\(^8\)Doing this is not possible if the conditions cannot be formulated as expressions that have a service object as their target (see also section 3.13.10).
11.3. Using the LinC language

Task preconditions vs. exceptions

When designing contracts, it is not always easy to decide if some requirement should be treated as a task precondition or if an exception should be thrown by the task if the requirement is not met. One argument that may be used in favour of one or the other solution, is if the condition is the sole responsibility of the requester of the task, and/or if not meeting the requirement is something the provider should know, and/or if the protocol flow should change if the requirement is not met. In the former case, the requirement should be stated as a precondition, while in the latter cases an exception should be used. There might be additional reasons, like the flexibility to be offered to the client, to decide on a particular way of handling requirements. Error handling strategies will be discussed in more detail below.

Error handling and actor responsibility

The palletIntake contract of section 7.4.1 provided a good example of how possible errors might be handled: even though the client manager assigned a platform to the truck driver (the human actor assisting the formal actor of this contract), the driver might make a mistake and drive the truck to the wrong platform. There are several ways to look at errors like this:

1. They are the responsibility of the client: this is expressed by stating the client's responsibilities as preconditions to the provider tasks. If a precondition is violated, the client is notified that some condition is violated; it then has to find out what went wrong and correct the situation. The provider is not even aware of any problem, it just waits until a valid task invocation message comes through.

2. They are the responsibility of the provider: this would be expressed by explicitly allowing the client to make a certain mistake (e.g., by removing preconditions), knowing that the provider would be able to cope with it. This method would only work if the mistake would not interfere with other contracts or with the remainder of the current contract!

3. The mistake is anticipated in the contract, and handling of the mistake is explicitly specified, most probably by an exception handler.

In the case of palletIntake, the first solution is the least complicated one, at least from the contract writer's perspective. The same service variable that is used by the provider to tell the client which platform to use (theAssignedPlatform), is used by the client in all subsequent task invocation messages. The first of these messages is unloadTruck(), which is probably sent as a result of the driver scanning the truck at the assigned platform. Both the scanned truck ID and the platform ID are passed to the provider; the latter argument is probably generated by scanner hardware located at the platform entrance. If the truck is scanned at the wrong platform, the value passed for thePlatform will not match the
previously assigned value (in `usePlatform ()`), and a `ProtocolViolation` exception will be generated. The scanner hardware might respond to this exception by flashing a red light to the driver instead of a green one, leaving it up to the driver to find out what went wrong (probably by calling the docking station controller). The driver must then move the truck to the proper platform. Following that, everything would be as usual. No special exception handling is needed, but the system could have been more helpful to the poor driver.

As not all driver errors will lead to problems (the other platform might be free for use), the above solution might be considered too restrictive. If the contract were to use a different platform variable for all but the `usePlatform ()` task invocation message, any errors would automatically become the provider's problem. The truck could effectively register at any platform, even at a platform that is already in use. By including a precondition on `unloadTruck ()`, viz. `pre thePlatform.isFree ()`, the provider can prevent the latter problem, while being as helpful as possible in all other cases. The client now only gets a `PreconditionViolation` if the platform is actually unavailable; the contract can continue normally if it is available. It is the provider's responsibility to allow for the client to select a different platform than the one assigned to him and to notify its own process logic of the new situation if this happens. This would complicate the provider's process logic, of course. It is still the client's responsibility to take action if the precondition were violated.

Taking one step further, the precondition on `unloadTruck ()` could be dropped in favour of an exception being generated if the driver went to the wrong platform. Since the check is no longer performed before the task is started, the provider could effectively delay the exception, or decide not to throw it at all! Delaying the exception would make sense if there were only few pallets at the platform waiting to be stacked; the client that made the mistake could be forced to wait until the platform is available. In very rare cases (e.g., if the previous truck load was owned by the same client), the provider might even allow the new truck to start unloading even though pallets from the previous load are still present. In this case no exception would be thrown at all. In case the provider decides to throw an exception right away, the situation is similar to precondition checking, as the only difference would be that the client wouldn't have to find out what happened; this is clearly conveyed by the exception itself. Of course, the provider's responsibilities are even greater than before, and the client's responsibilities would be limited to handling the exception.
Appendix A

Introduction to Java RMI and Jini

This appendix gives a short introduction to Java RMI and Jini, which together form the basis of the prototype LiNC implementation. Readers who are familiar with Java RMI and Jini may prefer to just have a quick glance; for others it provides a basis to understand chapter 9, which will show how these mechanisms are used to implement LiNC implementation. Readers who are new to distributed computing are advised to read chapter 8 first.

Since the LiNC platform relies on Java RMI, we will first provide a short introduction to this communication mechanism (section A.1). Following that, we will provide an introduction to the naming and lookup services underlying the implementation of the Logistic Contract Space; these services are built on top of Jini and its lookup service (section A.2).

A.1 Introduction to Java RMI

Java RMI [RMI, Gro01] is a Java specific implementation of distributed objects. RMI is like other distributed objects (e.g., CORBA [OMG02a]) in that it supports method calls between objects that reside on different machines on a network. It is different from other distributed object implementations in that it exploits the benefit of having a specific virtual machine that can execute Java bytecode on any platform that supports a JVM ([LY99]), thus opening the way to the transfer of code as part of a method invocation. As a result, any data passed between distributed objects can carry along the class implementations corresponding to that data. Of course, the interfaces implemented by these classes must be known or else the receiver cannot use the new classes (although a Java mechanism called introspection might come to help).

In RMI, a server can export object references to other machines. Such exported objects are called remote objects. Remote objects must implement the Remote interface; most of them also extend the UnicastRemoteObject class. Clients that hold a reference to a remote object can invoke methods on such an object; such remote calls are forwarded to the JVM that actually contains the remote object. Any non-remote parameters (including the references to their defining classes!) are serialized and sent as part of the remote method invocation. Any function results are serialized and sent back to the original caller. This
serialization of method arguments and results implies that arguments are passed by value; this is different from the mechanism of passing arguments between objects residing in the same virtual machine\(^1\). It is important to note that remote method invocations are synchronous: the caller is blocked until the remote object returns control (and possibly a result).

The call forwarding is largely invisible to the client, which appears to talk to a local object. There is one difference with local object calls though: method calls to remote objects may throw a RemoteException, and the client must be prepared to handle those. Of course, for the call forwarding mechanism to be invisible, such calls must be processed to include the extra code that is required by the 'remoteness' of an object. The RMI-'compiler' (rmic) takes care of this: when given a class that implements Remote it generates a new class whose instances act as proxies for the remote object; when used by clients, such proxy objects forward any method invocations to the remote object. The users of the proxy instance do not have to know where to find the server class: such details are delegated to the proxy, which contains the necessary code (generated by rmic) to locate the server.

### A.2 Introduction to Jini

Jini [AOS+99, Edw00, OW00] is a platform for distributed applications designed by Sun Microsystems. Sun's goal was to design a software platform for communication between hardware devices that can be connected to a network, like computers, scanners, printers, digital camera's, etc. The system would allow such devices to be connected to and disconnected from the network without any need for human intervention. Driver software would be downloaded automatically, and no user intervention would be needed to enable new devices to access other network resources or be accessed by the latter: the devices themselves would notify other devices of their presence and search the network for devices whose services they need in turn. As an example, a notebook computer that is plugged into the network would be able to find any printers available to it. If the printer should be unavailable at any time (because it ran out of paper, maybe) the client would then locate another printer automatically. The Jini architecture differs from conventional device drivers, as devices carry their own drivers within them, ready to be downloaded by clients. Jini device drivers are written in Java, and thus are platform independent: any hardware device that can run a Java Virtual machine can be a service provider or a service client. It was immediately recognized that the system was not at all limited to hardware devices. In fact, the same platform can be used to simplify any network based application, be it hardware or software, and the Jini platform is actually concerned with providing services on a network [Edw00, OW00]. Applying Jini as the basis for logistic programming like described in this thesis is merely stretching the original application area a bit further; this becomes particularly evident if we have a closer look at the Jini design goals as described

\(^1\)Actually, method arguments and function results are passed as Marshalled objects. Marshalled objects are serialized objects that carry a reference to the class they are an instance of, so the receiver can download the class code into the receiving virtual machine, if necessary.
A.2. *Introduction to Jini*

in [AOS+99] (AR1.1, pages 61-62):

"The focus of the system is to make the network a more dynamic entity that better reflects the dynamic nature of the workgroup by enabling the ability to add and delete services flexibly. ... End goals span different audiences; they include [but are not limited to]² the following:

- enabling users to share services and resources over the network,
- providing users easy access to resources anywhere on the network while allowing the network location of the user [i.e., the service client as well as the service provider] to change,
- simplifying the task of building, maintaining, and altering a network of devices, software, and users."

All these goals are very similar indeed to what we laid out as the main design goals of the LINC platform (see also chapter 3). It suffices to interpret the references to resources and services as references to *logistic* resources and services. To conclude, [AOS+99] presents some remarks about the Jini platform:

"A Jini system consists of the following parts:

- a set of components that provides an infrastructure for federating services in a distributed system,
- a programming model that supports and encourages the production of reliable distributed services,
- services that can be made part of a federated Jini system and that offer functionality to any other member of the federation."

Most of these remarks, the last one in particular, can be transferred to the LINC platform without change. The following chapter will show in some detail how Java RMI and the Jini services can be used as a basis for LINC.

Note that Jini is not a replacement for RMI; it is an addition to it (and more than that). The main differences between RMI and other distributed object registries can be summarized as follows (see also [OW00]):

- RMI and other client/server models require the client to know the exact network location of the server to be able to connect to the latter. Jini clients use a process called discovery to locate lookup services that they can use to locate the service they need. This process is more dynamic than more traditional naming and directory services.

²The bracketed parts were added by us.
• The Jini service is protocol independent, since providers can store any type of object (as long as it implements Serializable). The stored objects can use any kind of protocol to communicate to the service, e.g., the Object Management Group’s Internet Inter-Operability Protocol (IIOP) [OMG02a].

• Jini services are looked up on the basis of their type. The service types are standardized by the Jini Community Process, and providers are guaranteed to implement the type that they advertise. Known providers can also be contacted directly.

• Regular client/server implementations are not robust per se: in case of a network failure, the communication is likely to ‘hang’ or to fail with timeout errors. Robustness can only be obtained by adding extra layers to the basic network protocol layer. Jini addresses both these issues by providing a way to locate clients and/or servers dynamically, to add the possibility of redundancy to Jini-based services, and by providing a basic mechanism to detect and recover from network errors.

A.2.1 The lookup service

If devices are to locate each other without human intervention, there must be some way for devices to advertise themselves and to locate the services they need. The Jini lookup service is used for this. The lookup service is itself a Jini service that can run on a potentially unlimited number of machines in a network. Applications can locate all these lookup services, and register with each of them. Applications can also listen for new lookup services, and register with them whenever they appear. When an application registers, it has to upload its Java signature and any other information that might be useful to distinguish the application from similar services. E.g., a printer might want to include its location in the registered information, to prevent users in different buildings (or even worse: in the competition’s building) from using them. Related sets of services on a network are often referred to as Jini communities. Jini lookup services are organized into groups that correspond to those communities: these groups are specified when instances of the lookup service are created. Communities and groups are very similar to the LINC service domains, and can be used as the basis of their implementation.

The lookup service is robust because its resources are leased by its users. These leases must be renewed on a regular basis, or the advertised service will be removed from the lookup service. As a result a service that is disconnected will be removed eventually. The same will happen if some part of the network fails, making the service inaccessible, or if a device is struck by a power outage. Clients can always use the lookup service to find other instances of the service they need, even through other lookup services if necessary. Since the lookup service runs on several machines in the network, the system is also protected against a lookup service failing occasionally. It is this mechanism that forms the basis of the self-healing property of Jini communities.

When a client is looking for some service, it locates any lookup services close to it, and sends them a signature (i.e., Java interface) of the service it is looking for. If a matching
service is found, the lookup service will respond by sending the actual object implementing the requested interface. Sometimes the returned object implements the requested service all by itself, but more often it implements a proxy that will forward any calls over the network to a remote object implementing the service. This remote object is the actual provider of the service. Figure A.1 shows how the process works.

![Diagram of Jini lookup service]

Figure A.1: Using the Jini lookup service

Most Java remote objects are implemented using RMI (see section A.1), so most Jini enabled applications use this network protocol for the communication between distributed objects. This is by no means the only way though, so application designers might choose other network protocols to implement communication between remote objects. Having said that, we should also mention that some parts of the Jini platform (e.g., Jini distributed events, some Jini transaction services, and the Sun-supplied implementation of the lookup service) depend on RMI.

A.2.2 A simple Jini application

A typical Jini application consists of a set of client-side interfaces and classes, a set of provider-side interfaces and classes, and a set of interfaces that are shared between the client and the provider. For the purpose of this explanation, we will adopt the architecture presented in [OW00]; below we will introduce some changes to this basic architecture that makes it better suited to the LINC context.

Starting with the shared interfaces, there is an interface representing the service. In the absence of a specific service name, we will always use ‘Service’ to denote the actual service name. Whenever we refer to a specific service name (and in actual code, of course) all
occurrences of Service will be substituted by the actual service name). We'll come to the second shared interface in a few moments. The Service interface is neither Serializable nor Remote; the client can get it from anywhere or just create it himself. The same is true for the provider. At the provider side, the Service interface is extended to the ServiceProxy interface, which also extends Remote. It is the latter interface which will be registered with the lookup service by the instance of the main provider class during startup of the latter. See figure A.2 for the various classes and interfaces that were just mentioned.

![Diagram of Jini service architecture](image)

**Figure A.2: A typical Jini service**

The ServiceProxy interface is implemented by the main provider side class, ServiceProxyImplementation. Usually, this class extends UnicastRemoteObject (not shown in figure A.2). When the ServiceProxyImplementation is instantiated, it registers itself with the lookup service.

Typically, neither the Service interface nor the registered ServiceProxy interface contain any service specific methods. Instead they define a single getInstance() method that allows the client to retrieve an object implementing the second shared interface: ServiceRegistration. It is this second interface that fully specifies the service offered by the provider. At the provider side, this ServiceRegistration interface is implemented by the ServiceRegistrationImplementation class, which also implements Serializable.

Now what happens when a client is looking for a certain service is the following: it asks the Lookup Service for all providers implementing the Service interface. In response, the client will get one or more references to instances of ServiceProxyImplementation. It then selects a particular provider and asks it for an instance of the ServiceRegistration interface. In

---

Note that this is just the typical Jini application; the separation between the published Service interface and the provider side ServiceProxy interface was in fact introduced in order to allow providers to use communication mechanisms other than RMI. In that case the ServiceProxy interface would probably extend some other protocol specific interface instead of Remote.
return, it will receive a serialized instance of the ServiceRegistrationImplementation class. This instance will forward any communication to the remote instance of the ServiceProxyImplementation class. The process has been visualized in figure A.1.

The provider/provider registration distinction

The need to use a separate serializable registration class (instead of a generated stub) stems from Jini licensing mechanism, which requires the client to hold a local (!) license object for the service. The ServiceRegistrationImplementation instance returned to the client contains such a a local license object (given to it by the provider when it created the instance), and sends it to the provider as part of every forwarded method invocation. All these licensing issues are invisible to the client though, who just uses the local instance of the ServiceRegistration class that it has received from the provider.

A further advantage of having a non-remote registrar interface is that the implementation of the latter may use any protocol to communicate with the provider, not just RMI. Neither of the shared interfaces has any dependence on RMI.

Keeping the client state

The architecture described above defines a single provider instance that must (in principle, at least) be able to handle more than one client simultaneously\(^4\). Most interesting services span more than a single interaction, though. Since all RMI calls are independent of each other, no matter if they come from a single client or from many different ones (the provider just fires a new thread to execute a particular incoming call), something special needs to be done to preserve the state of each client’s communication. Generally, the client’s license object (which is passed in each method call anyway) is used by the provider as a key to retrieve the state for that client.

While the mechanism just sketched is not a bad solution, there is another way that might be a little more useful under some circumstances (at least it is adopted in the LiNC implementation!). The first step to understanding the alternative solution, is to see that the state-keeping ‘problem’ is caused by the fact that a single instance of the ServiceProxyImplementation is to handle all clients! If the actual handling of each client were delegated to a new provider instance, that instance itself could keep the interaction state for that client. This architecture is depicted in figure A.3; note that our naming scheme is a little different from the naming scheme that was used in figure A.2.

The main difference with the previous system is that the new provider’s getInstance () method not only creates an instance of the service registration implementation class (renamed to ServiceProxyImplementation here), but also an instance of the new RemoteSer-

\(^4\)If the provider can only handle one client, its getInstance () method might keep a queue of waiting clients and block until all preceding clients have been handled; alternatively, getInstance () might throw an exception if the client can’t be immediately served, or if it can’t be served within a certain period.
Figure A.3: An alternative Jini service

viceRegistrationImplementation class. The former instance is returned to the client, while a reference to the latter instance is embedded within the former: when instantiating the service proxy instance, the provider not only passes the lease object for the particular client, but also a reference to the new registration implementation instance that is to handle that client. Since the latter instance handles exactly one client it can simply keep the state of that client without having to retrieve it.
Appendix B

Glossary of terms

The following glossary of terms provides short definitions for the most frequently used terms in this thesis. Many definitions will refer to glossary terms themselves; such terms will be marked in the usual way, i.e., using italic typeface, and followed by an asterisk.

2-party contract: A service contract* in which there are 2 parties*, one playing the role of the provider* and the other playing the role of the client*.

actor: An actor is a software object representing a party in a logistic operation. Some authors use the word ‘agent’ instead of actor; the term ‘actor’ is preferred over ‘agent’ because the latter term carries some additional (AI-related) connotations that we want to avoid in the current context: while actors may exhibit ‘intelligent’ behaviour, this is not implied or even necessary.

actor signature: The actor signature is a description of the skills* of a logistic actor*.
Skills may be invoked by other parties in the service contract*; a skill that is invoked becomes a task*. Individual skills/tasks are defined by means of a task signature*; an actor signature is therefore a collection of task signatures.
Every logistic actor corresponds to a Java interface in the Logistic Contract Space*; the task signatures that are listed in the actor’s signature correspond to public methods in this Java interface. When a contract is executed, the objects (i.e., class instances) that correspond to the logistic actors of the contract must of course be instances of those same Java interfaces.

agent: An independent software entity that is capable to perform some useful task; usually, an agent will communicate and cooperate with other agents. The concept of an agent evolved as part of AI research and still carries a connotation of ‘intelligent’ behaviour. To avoid this undesirable semantics, we prefer to use the word actor* instead.

Actor Interaction Modeling (AIM): A design/programming paradigm that focuses on describing the interactions between (remote*) software objects. The interactions are described by a service contract* containing the objects’ signatures* as well as a protocol*. The protocol describes how the objects are supposed to interact. This
approach is especially useful in a distributed environment, where the various objects are not implemented by the same party. It also helps maintenance and evolution of software objects. In the context of LINC, the software objects correspond to logistic actors*.

**ambiguous, ambiguity:** Ambiguity is a problem that sometimes occurs while executing a protocol*. If a given task invocation message* or subcontracted interaction pattern* could continue execution in more than one possible 'path' through the protocol, the protocol is called ambiguous. As it would be impossible to know how to continue the protocol, ambiguous interactions will be rejected and a AmbiguousProtocol exception will be thrown.

**applet:** Usually small application program that can be executed from within a web-browser.

**asterisk:** The symbol used to mark terms that are explained in the glossary.

**bound:** One of the two possible states of a logic variable*. A logic variable is said to be bound if a value has been assigned to it.

**charge:** A (currently implicit) task* that must be executed by all actors* of a service contract* and signaling the start of that contract. No party is allowed to start executing the protocol* section of the contract until all parties have executed the charge (). Charging the contract also has a legal meaning: after a party has executed its charge (), it has committed itself to performing its role in the contract. If it does not live up to the obligations associated with that role, legal action may follow.

**client:** A logistic actor* that intends to use some service* implemented by a provider*.

**client signature:** The actor signature* of the client*.

**community, — Java community:** Open organization of (Java) users which formalizes the decision process that guides evolution of the Java platform. A similar organization was created to coordinate evolution of the Jini platform.

**composite node:** A journal tree* node that has daughter nodes corresponding to nested interaction patterns*.

**contract:** See service contract.

**contract signature:** See service signature.

**contractual exception:** An exception that is defined and handled within a service contract*. All contractual exceptions are defined as part of a task signature* and have to be handled within a try construct*. Handling of contractual exceptions can be formal or informal. 'Formal' means that the exception handler* is defined using regular interaction patterns* that are to be followed if the exception occurred. 'Informal'
means that the handling is not defined within the service contract*; in that case the error must be handled by means of ad-hoc involvement of one or more actors of the contract. An exception handler must either be specified as a formal handler* or as an informal handler*; a combination is not possible.

**deep copying**: When copying an object, we need to distinguish between **deep copying** and **shallow copying**. If a deep copy of an object is made, all further objects that are referred by that object are also deep-copied, and so on (recursively). Consequently, the original object structure and the new object structure are completely disjunct. Care should be taken to avoid unnecessary structure duplication if the original object structure contained multiple references to a shared substructure, in particular when the shared structure represents a cycle. Shallow copying, in contrast, only copies the original object; all references from that object will be shared between the original object and its copy.

**defensive copying**: Since Java does not provide a way to mark objects as immutable*, it is often necessary to make a copy of an object before it is passed as a parameter in a **task invocation message** to an actor. Even though this doesn’t prevent the actor from changing the value, it does protect other parties from being influenced by the change. The copying code is inserted by the LINC compiler; implementors of the corresponding classes have to make sure that all copies are **deep copies**.*

**delegation**: Parties may ask 3rd parties to execute (parts of) a **task** on their behalf, usually be instantiating a **service contract** with those parties as a **sub-contract**. This process, which is usually described inside a **process block**, is referred to as delegation. The instantiation process itself is referred to as **subcontracting**.

**delegation contract**: The term ‘delegation contract’ was used in some older publications ([ELL99, ELL00]) to refer a form of contract that integrated delegation of services into the framework of logistic contracts. In the current language, delegation of services is considered a form of **subcontracting** that takes place in (client or provider) **process blocks**.

**discharge**: A (currently implicit) **task** that must be executed by all actors* of a service contract* and that signals the end of that contract. No party is allowed to withdraw until all parties have executed the **discharge**. Of course, a party will only discharge if it thinks that all parties have fulfilled their obligations. Discharging the contract also has a legal meaning: after all parties have executed their respective **discharge**, the responsibility of any individual party has ended.

**domain**: See **service domain**.

**enterprise system**: The term **enterprise system** is used to denote software applications that cover the most important aspects of an enterprise. Depending on the field in which the enterprise is active, the system might include **supply chain management**, **order processing**, **inventory management**, **project management**, **personnel management** etc. Enterprise systems tend to be large, and because of that, are often inflexible.
exception handler: A protocol* construct that prescribes the interactions that have to take place after an exception has been thrown. Two kinds of exception handler are distinguished, i.e., formal handlers* and informal handlers*. Usually, an exception handler does not do anything by itself; it merely expects the actors* to handle the error. There is one exception, though: an informal handler may specify that a special exception handler task* is to be called by the protocol verifier* itself.

exception handler task: A task* that is intended to handle the erroneous situation that results from an exception being thrown. Often, exception handler tasks are implemented by all actors* of a service contract*. Currently, exception handler tasks are not distinguished from regular tasks, even though there are some extra requirements: in particular, a handler task is not allowed to have a precondition*; a postcondition* is recommended. The LINC compiler checks that these requirements are met.

formal exception: Exception that is handled by executing a well defined protocol* fragment to restore a workable state. The handler* for a formal exception usually specifies an alternative protocol to be followed after the exception was thrown.

formal exception handler: Exception handler* for a formal exception*.

general exception: A general exception is an exception that is not named in any task signature*. It can be thrown by any task, and sometimes even by the protocol verifier* itself. Contrary to named exceptions*, general exceptions need not be handled and can be propagated to the context of the service contract* in which they are thrown. Some of the most common general exceptions are ProcessError and TimingError.

handler (task): See exception handler task*.

immutable objects: Objects that cannot change after they have been created. Immutability is an important attribute of objects, especially in the context of distributed software. Unfortunately, the underlying Java platform does not provide language support for (im)mutability.

index argument: A designated argument in a task invocation message* that is used to pass multiple indexes and/or service indexes. Index arguments serve to distinguish between branches in a concurrent multiple construct.

informal exception: Exception that cannot be handled within the service contract*. Contrary to what happens with a formal exception*, the handler for an informal exception does not usually specify an alternative protocol. Instead, it invokes a task, often on more than one party*, to inform them of the exception's occurrence. The parties are expected to cooperate and restore a workable state and continue from there. The details of the interaction that is necessary to achieve this are not described though.

informal handler: Exception handler for an informal exception*.
interaction pattern: The protocol* section of a service contract* consists of interaction patterns. Every interaction pattern is either a simple interaction pattern or a composite interaction pattern. Simple interaction patterns describe basic interactions such as sending a task invocation message* or instantiating a sub-contract*. Composite interaction patterns are used to describe e.g., the timing relationships between two or more simple or composite interaction patterns.

italic: The typeface that is used to print glossary terms. Besides from being printed in italic, glossary terms are followed by an asterisk*.

journal (tree): Execution of a service contract* can be logged in the form of a journal tree. Each node of this tree corresponds to a protocol construct*. If specified in its task signature*, a task* may also put logging information into the node that results from execution of that task. Tools are foreseen to browse the journal tree. The journal tree can also be used as input for the report generator specified in the report section of the service contract.

journal value: A value that may be inserted into the journal tree* by an actor* upon termination of a task*.

logic variable: A Prolog-style variable [CM81] that can be in two states: unbound* and bound*. If a value is assigned to an unbound logic variable, the variable becomes bound to that value. If a value is assigned to a variable that is already bound, the assignment succeeds if the value it is bound to is equal to the newly assigned value. If the values differ, the assignment fails.

logistic actor: A participant (party) in a service contract*. Usually, parties implement real-world processes in the logistic chain. Examples may be air carriers, shipping companies, and courier services, but also automated installations like AGV’s or quay cranes. Humans may also become actors in a service contract if a suitable user interface is available. All logistic actors are defined by classes that must be defined in the Logistic Contract Space*; such classes must derive from LincBasicActor.

Logistic Contract Space: The Logistic Contract Space is where all compiled service contracts* are stored, available for download by actors*. Often, service contracts are uploaded by providers and downloaded by clients; more elaborate procedures are to be followed in case of multi-party contracts. Besides service contracts, the Logistic Contract Space contains Java interfaces and classes for the types that can be used in contracts, and a number of basic logistic actor* types.

multi-party contract: Generalization of a 2-party contract, in which there are n parties instead of just 2. The roles of the provider* and the client* are not distinguished any longer: all parties involved are (just) logistic actors*. The additional actors can e.g. be used to specify parties to which services can be delegated*, or parties that need to monitor the service while it is being executed. Examples of the latter are a so-called patron that is responsible for the well-being of the service object*, customs, and insurance companies.
**multi-tiered system:** Modern software design techniques advocate the design of multi-tiered systems. At the lowest layer, such system contains (relatively) stand-alone applications or functions. These functions are either very general, like a database system, or very specific, like a system that models specific business processes. As the database system also is complemented by business specific database schemes, we can refer to the systems in this layer as the *business logic.* In the top layer one can find the applications to which users interact directly. These applications might be designed for data entry, data display, system control, etc. To achieve a high level of flexibility, these applications are designed independently of the business logic mentioned above. The intermediate layers constitute a form of ‘binding’ between the ‘user interfaces’ of the top layer and the business logic of the bottom layer. Note that the term ‘user interface’ is used loosely, since the user might well be a computer application itself. A system that is designed according to these principles is very flexible indeed. It is easy to change user interfaces, to combine existing user interface functions into new ones without having to change the business logic. Likewise, the business logic can be re-implemented (e.g., to make it more efficient or adapt it to new hardware or software) without changes to the user interfaces.

**named exception:** A named exception is an exception that is declared in a *task signature*. Named exceptions must always be handled (by means of a *try-construct*) in the *protocol*, usually by a *formal handler*.

**overloaded, overloading:** Names are said to be *overloaded* if they can refer to more than one thing. In the LINC language, overloading is only allowed with regard to *task* names. Also, overloading is only allowed if it is always possible to use context to determine the intended meaning; this means that the numbers of arguments of the task, or their types, must be different.

**contract party:** The real-world logistic actor that is behind a *logistic actor* as defined by a *service contract*.

**postcondition:** A *skill* may have a postcondition specified in its *task signature*. A postcondition is a condition that must be fulfilled after execution the skill (as a *task*) has been completed. Since terminating a task would be meaningless if the postcondition is violated, the *protocol verifier* intercepts such violations.

**precondition:** A *skill* may have a precondition specified in its *task signature*. A precondition is a condition that must be fulfilled before the skill can be invoked. Since executing the skill (as a *task*) would be meaningless if the precondition is violated, the *protocol verifier* intercepts such violations.

**process block:** A formalized *comment* that provides a (partial) description of a *actor’s* process logic (also called business logic). Process blocks are often used to show that part of the implementation of a task is *subcontracted* to a 3rd party. At a later stage, *protocol* checking will probably be extended to include the interactions that take place under control of sub-contracts, even if these occur in a process block. Such
checking will probably be limited to the propagation of general exceptions* from the subcontracted service.

**protocol**: The protocol is the most important part of the service contract*. It describes how the contract parties are supposed to interact during execution of the contract. During execution of the contract, all interactions will be verified against the protocol. If a party tries to invoke a task that is not allowed given the state of the protocol, the action will be blocked and a ProtocolViolation exception will be thrown to the offending party. The sender of the erroneous action can then try to correct the situation, and the other contract party* will not even know that a problem has ever occurred. As a result, the handling of such errors can be kept strictly local.

**protocol construct**: Any one of the interaction patterns* that can be used to define a protocol*.

**protocol event**: An event that changes the state* of a protocol. The only protocol events are the starting and ending of a task*.

**protocol state**: The state of execution of a protocol*; this state determines which interactions can legally follow. Every interaction corresponds to at least two protocol events* and causes the state of the protocol to be changed. It is important to note that the state is is a complex entity that might not be clearly recognizable to the observer: it is implicitly defined by the set of all interactions that may occur in that state. If the state transition resulting from a particular interaction cannot be not uniquely determined, the protocol is ambiguous*. In addition to the state of the protocol itself, the individual interaction patterns making up the contract also have a state, though not every aspect of this state applies to every construct:

- Constructs are called active if tasks are currently executing under them, inactive if no task is currently executing under them and if it contains tasks that have not been started yet, or completed if no task are currently being executed under them and there are no tasks left to be executed under them. This rule applies to all protocol constructs, with the restriction that tasks can only be active or completed.
- Furthermore, constructs are called open if they contain nested constructs that have not started yet, and closed if there are no nested constructs left to be started¹. This rule does not apply to task invocation messages.

**protocol verifier**: The run-time object that checks if the interactions between the actors* conform to the protocol* and that logs all interactions in the journal tree*.

**provider**: A logistic actor that has to offer some service to clients.

**provider signature**: The actor signature of the provider.

---

¹While the Active/Inactive/Completed distinction applies to the entire (sub-)protocol headed by some construct, the Open/Closed distinction only applies to the construct itself!
proxy: A local placeholder for a remote object*. This placeholder implements the same interface as the remote object, but implements this interface by forwarding all method calls to the remote object. The results of executing remote methods are returned to the proxy object, which in turn returns them to the original caller. When necessary, the proxy can enforce security measures like authentication and encryption.

root location: — of the Logistic Contract Space: the directory (or set of directories) in which the Logistic Contract Space is located. The Logistic Contract Space can be distributed over a set of directories, in which case a search path is used to locate the various service domains* and the entities within those domains. When compiling code generated by the LINC compiler, the root(s) of the Logistic Contract Space must be part of the Java compiler’s class path.

remote objects, — types: Classes that implement the Java Remote interface, or objects that are instances of such classes. Service contracts* describe distributed applications almost by definition. The platform adopted by the current LINC implementation is Java RMI. On this platform, objects passed between remote parties are either remote or serialized, i.e., the corresponding classes must implement either the java.rmi.Remote or the java.io.Serializable interface. Remote objects could reside at any machine; most often this will be the machine at which one of the contract parties resides. There is only one copy of the object, accessible to all parties that get hold of its (remote) reference.

Serializable types: Types that implement the Java Serializable interface. Service contracts* describe distributed applications almost by definition. The platform adopted by the current LINC implementation is Java RMI. On this platform, objects passed between remote parties are either remote or serialized, i.e., the corresponding classes must implement either the java.rmi.Remote or the java.io.Serializable interface. Serialized objects can reside at many machines at any time: when passed from one machine to another, the receiver will get a copy of the object instead of a reference. One must beware not to change the copy, since in that case the different instances would get out-of-sync.

service: A real-world service to be executed by a number of logistic actors*.

service contract: The formal description of a service*, i.e., of the various tasks* that may be executed by the contract parties in order to execute the service. Note that the internals of these tasks are left unspecified; this is the business logic that each party must implement in order to have its tasks executed by the other parties. The task’s business logic can implement anything, as long as the pre- and postconditions* of the tasks are not violated.

The service contract can be compiled by the LINC compiler, which then generates a set of Java interfaces and classes for the various contract parties. For each party, a skeleton will be generated that must be completed to implement that party’s business logic. After having been compiled by the Java compiler, the various parties will
communicate over the network using Java RMI. Jini is used to make the parties locate each other in the first place.

**service domain:** A part of the *Logistic Contract Space* that represent a specific application area. Domains should be defined by the LINC user community.

**service object:** An entity that is affected by a *service*; often implemented as a Java object or a kind of 'database' representing the state of the affected entity. The state will be updated by the protocol verifier to reflect the state of execution of the service.

**service signature:** To help prospective clients to select a suitable *service contract*, every contract is characterized by signature that is unique to that contract. The service signature specifies a name for the service and for the domain that the service belongs to. The service signature is complemented by a set of *actor signatures* and by a section specifying the *service objects* operated upon.

**signature:** There are three kinds of signature in a service contract: the *service signature*, the *actor signatures*, and the various *task signatures*. See the various entries for these terms for more information.

**skills:** A piece of work that can be done by an *actor* in a monolithic way, i.e., without requiring further interaction with the requestor. A *client* can ask an actor to apply one of its skills. The execution of a skill is called a *task*; the request itself is therefore called a *task invocation message*. Each skill is defined by means of a *task signature*.

**state:** See *protocol state*.

**subcontracting:** The instantiation of a *service contract* as part of another contract. If a *service* is subcontracted inside a *process block*, we speak of *delegation*.

**task:** The execution of a *skill*. Tasks can behave like regular (Java) methods, and may take time to execute; they can also be made to behave like messages that communicate data to other *actors*. In the latter case the time of their execution can be ignored. Such message tasks are currently not marked as such, but they generally do not have a *postcondition* and/or a *journal value*.

**task invocation message:** The most basic of all interaction patterns, that consists of a request to a contract party to execute a *task*. Execution is *synchronous*, i.e., the task invocation message does not terminate until after the task has been completed.

**task signature:** The *tasks* provided by a *logistic actor* are specified by means of task signatures giving the name and arguments of the tasks. In addition, a task may have *pre- and postconditions*, may throw *named exceptions*, and may specify a *journal value* to be included in the *journal tree*.

**time unit constant:** A constant defined in the *Logistic Contract Space* to denote a specific amount of time, e.g., *Year* or *Minute*. Time unit constants can be used in timed interaction patterns.
unbound: One of the two possible states of a logic variable*. A logic variable is said to be unbound if no value has been assigned to it.

wrapper type: A reference type that wraps a primitive non-reference type. Wrapper types are used in LINC to eliminate the distinction between primitive and non-primitive types.
Bibliography


Bibliography


Towards on-line logistics: the LinC contracting language

Logistic systems are evolving from static, ERP-based enterprise systems to highly dynamic, network based ‘virtual enterprises’ that are established to execute a task at hand and are dissolved after this task has been completed. Special mechanisms are needed to support the formation of such virtual logistic enterprises, to coordinate the tasks that are delegated to the individual logistic actors, to validate and protect the interactions between these actors, and to record the progress being made in executing the task, both for practical and for legal reasons. Apart from being able to handle regular situations, these mechanisms must be able to handle the exceptional situation that will arise due to e.g., (mechanical) failures, (human) errors, and delays.

This thesis proposes logistic contracts as a means to achieve the goals that were implied above: enterprise formation is supported since actors can advertise their skills by listing the contracts they support and the roles they can play in them; coordination, validation, and protection is supported by using contracts to guide the interactions during runtime; and finally, contracts allow interactions to be recorded in a journal that reflects the logistic design. The logistic design method is ‘implemented’ by a logistic design language, called LinC (Logistic Interaction Contracting). The language allows designers to specify the logistic actors’ signatures as well as an interaction protocol that is used to verify all attempted interactions during run-time. The design of this contracting language has been the main research topic. The language design process has been documented by including a detailed rationale for many constructs as part of the language description.

The language was validated in two ways: by using it to perform a number of case studies, and by implementing a prototype compiler and contract verifier. Two complete case studies will be presented here: the first one is concerned with the transport of valuables between two different banks, while the second one describes a complete cross-docking station that is used to (re)distribute goods between a number of producers and a number of consumers (supermarkets). The case studies show how the language can be put to use to design a medium-scale logistic system. At the same time, the case studies are annotated with remarks explaining how feedback from the cases has affected the language. The prototype implementation of a compiler/verifier is used to show that a protocol verification mechanism can indeed be implemented; it also provides a basis for supplementary tools to manage logistic contracts and to browse execution journals.

Job Honig
Naar on-line logistiek: de LinC contracteringstaal

Logisticke systemen maken een ontwikkeling door van statische, op ERP gebaseerde 'enterprise systems' naar zeer dynamische, netwerk-gebaseerde 'virtual enterprises' die worden opgezet om een bepaalde taak uit te voeren, en die weer worden afgebouwd wanneer die taak is voltooid. Er zijn speciale mechanismen nodig die de vorming van zulke virtuele logistieke ondernemingen ondersteunen, die de taken kunnen coördineren die worden gedelegeerd aan de individuele logistieke actoren, die de interacties tussen deze actoren kunnen valideren en beschermen, en die de voortgang die wordt gemaakt bij de uitvoering van een taak kunnen registreren, zowel met een juridisch doel als om praktische redenen. Deze mechanismen moeten niet alleen in staat zijn om verwachte situaties af te handelen, maar ook de onverwachte situaties die zouden kunnen ontstaan als gevolg van bijvoorbeeld mechanische problemen, (menselijke) fouten, en vertragingen.

Dit proefschrift introduceert logistieke contracten als middel om bovenstaande doelen te verwezenlijken: de vorming van virtuele ondernemingen wordt ondersteund doordat actoren hun diensten kunnen adverteren door een lijst aan te bieden van de contracten die zij ondersteunen samen met de rol die zij in deze contracten kunnen spelen. De coördinatie, validering, en bescherming van de interactie wordt ondersteund door deze contracten te gebruiken als leidraad voor de feitelijke run-time interactie. Tenslotte bieden contracten de mogelijkheid om de interactie te registreren in een 'jurnal' dat een afspiegeling vormt van het logistieke ontwerp zelf. De voorgestelde logistieke ontwerpmethode is 'geimplementeerd' in de vorm van een logistieke ontwerp-taal, LinC (Logistic Interaction Contracting) genaamd. Deze taal biedt ontwerpers de mogelijkheid om de 'signature' van een acteur te beschrijven (d.w.z., de verzameling taken die de acteur kan uitvoeren). Daarnaast kan een interactie-protocol worden gespecificeerd; dit protocol wordt gebruikt om de feitelijke interactie te verifiëren. Het ontwerp van deze 'contracting' taal vormde het kern van ons onderzoek. Het ontwerpproces is gedocumenteerd door de taalbeschrijving te voorzien van een gedetailleerd 'rationale' voor de meeste taalconstructies.

De taal is op twee manieren gevalideerd: enerzijds door haar toe te passen in een aantal casussen, anderzijds door een prototype te implementeren van een compiler en een contract checker. In dit proefschrift worden 2 case studies gepresenteerd: de eerste betreft het transport van waardepapieren tussen verschillende banken; de tweede study beschrijft een compleet 'cross-docking' systeem (een cross-docking systeem wordt gebruikt om tijdelijk goederen op te slaan die moeten worden getransporteerd tussen verschillende producenten en verschillende afnemers (supermarkten zijn een goed voorbeeld). De case studies laten zien hoe de taal kan worden gebruikt om een middelgroot logistiek systeem te specificeren. De case studies zijn geannoteerd met opmerkingen die toelichten hoe de cases het ontwerp van de taal hebben beïnvloed. De implementatie van een prototype compiler en checker laten zien dat het inderdaad mogelijk is om een protocol verificatie mechanisme te implementeren; het verschaf ook een basis voor de ontwikkeling van additionele gereedschappen voor het 'managen' van logistieke contracten en het inspecteren van executive-journaals.

Job Honig
About the author

Job Honig was born on September 4, 1953, in Gouda, The Netherlands. After completing Gymnasium β in Gouda, he studied Electrical Engineering at the Delft University of Technology (known as 'TH Delft' at the time) in 1972. He graduated 'met lof' in 1979.

As early as 1976, his main research interest was in the field of 'Natural Language Processing'. As a university employee since 1979, he participated in international projects more than once, and developed a series of comprehensive natural language processing systems, the latest system being based upon an Ada 95 implementation of the Double Dotted parsing algorithm [VH89, Hon94]. He stayed active in this field until mid 2000, when his department decided there was no future for Computational Linguistics at the Delft University.

Because of his interests in software engineering and programming language design, he then was asked to implement a compiler for a communication language for distributed objects. It is this work which lead to the design and implementation of a new language and compiler, the topic of the research described in this thesis. Unfortunately, circumstances at the Delft University are such that at this time it is highly uncertain if the work described in this thesis can be continued.

Apart from his research for TU Delft, the author has developed both commercial software applications and software courses. This work was done in the context of a one-man software engineering consultancy firm. More recently, he started to learn about digital photography and image editing (software); an interest that can very easily absorb all of the available resources (i.e., computing power, storage capacity, time as well as money). To its advantage, it should be mentioned that this field has a lot more depth than previously imagined.