Marco António de Castro Barbosa **Specification and Refinement of Software Connectors**

UMinho|2009

Universidade do Minho
Escola de Engenharia

Marco António de Castro Barbosa

Specification and Refinement of Software Connectors

Julho de 2009

Universidade do Minho
Escola de Engenharia

Marco António de Castro Barbosa

Specification and Refinement of Software Connectors

Tese de Doutoramento em Informática
Área de Conhecimento de Fundamentos da Computação

Trabalho efectuado sob a orientação do
Professor Doutor Luís Soares Barbosa

Julho de 2009

Contents

1 Introduction **1**

1.1 Motivation, objectives and research questions 1

1.2 Thesis structure and contributions 4

1.3 Context: Components, coordination and architectures 8

1.3.1 Components 8

1.3.2 Architectures 10

1.3.3 Coordination 12

1.4 Mathematical preliminaries 15

1.4.1 Functions 15

1.4.2 Relations 18

1.4.3 Coalgebras 21

2 Stateless Software Connectors **25**

2.1 Synchronous stateless connectors 25

2.2 New connectors from old 29

2.3 An example 33

2.4 Towards a connector calculus 35

2.4.1 The equational fragment 35

2.4.2 Connector refinement 40

2.5 Extensions 41

2.5.1 Time-stamped connectors 41

2.5.2 Connectors with alternative ports 42

3 Stateful Software Connectors **45**

3.1 Introduction 45

3.2 Connectors as coalgebras 47

3.2.1 The general model 47

3.3 Combinators 50

3.3.1 Concurrent composition 50

3.3.2 Interleaving 51

3.3.3 Hook 51

3.4 Examples 53

3.4.1 Broadcasters and Mergers 53

3.4.2 Drains 55

3.4.3 The Dining Philosophers 56

3.5 Towards a connector's calculus 58

3.5.1 Bisimilarity 58

3.5.2 Reasoning about connectors 61

3.6 Towards mobile connectors 65

3.6.1 Behaviour and configurations 67

3.6.2 A connector for orchestration 68

3.6.3 Coordination Patterns 70

3.6.4 An example 71

4 Behavioural Interfaces 75

4.1 Interfaces and behaviour 75

4.2 Revisiting process algebra 77

4.2.1 Introduction 77

4.2.2 Combinators 78

4.2.3 Interaction 80

4.3 Proofs and prototypes 81

4.3.1 Hiding 81

4.3.2 The expansion law 84

4.3.3 Functional prototyping 85

4.4 Interruption and Recovery 86

4.4.1 Apomorphisms 86

4.4.2 Parallel Composition with Interruption 87

4.4.3 Conditional Fusion 89

4.4.4 A Recovery Operator 92

4.5 Behaviour-annotated interfaces 93

5 Context-aware Connectors and Architectural Configurations 97

5.1 Interfaces, connectors and configurations 97
5.2 The coordination layer 101
5.2.1 Connectors 101
5.2.2 New connectors from old 105
5.3 Propagation of context dependent behaviour 112
5.4 Towards a connector calculus 115
5.5 Configurations 120
5.5.1 Examples 123

6 Case-study: Interactors 131
6.1 Introduction 131
6.2 A logic for behaviour 133
6.2.1 A modal language 133
6.2.2 Typical properties 136
6.3 $\mathbb{M}$-interactors 136
6.3.1 A language for $\mathbb{M}$-interactors 136
6.3.2 Composing interactors 138
6.4 The coordination layer 139
6.4.1 Elementary connectors. 140
6.4.2 New connectors from old 141
6.5 Configurations of $\mathbb{M}$-interactors 146
6.6 Concluding 149

7 A Functional Library for Prototyping Software Connectors 151
7.1 Functional components and their interfaces 151
7.2 Connectors 153
7.3 Combinators 157

8 Conclusions and Future Work 161
8.1 Concluding 161
8.2 Future work 163
8.3 Styles and performance in software architecture 165
8.3.1 Architectural styles 165
8.3.2 Performance in software architecture 171

List of Figures

2.1 The *External control flow* pattern. 34
2.2 A synchronization barrier 34

3.1 The *broadcaster* connector. 54
3.2 The *merger* connector. 55
3.3 Dining Philosophers (1). 57
3.4 A *Fork* connector (1). 57
3.5 Dining Philosophers (2). 58
3.6 The initial configuration. 71
3.7 The final configuration. 72

5.1 An example of a *configuration*. 99
5.2 A *merger*. 110
5.3 A *broadcaster*. 112
5.4 Imp — An '*impossible*' connector. 117
5.5 SB — A *synchronization barrier*. 118
5.6 XR — The *exclusive router* connector. 118
5.7 An *alternate merger*. 121
5.8 A elementary bank system 123
5.9 A *folder* from two stacks. 125
5.10 Air traffic control configuration 127
5.11 Air traffic control configuration - connector 128

6.1 A model-based Interactor. 132
6.2 A window interactor 138
6.3 The space interactor 139
6.4 A classical solution 140

6.5 A *merger*: $\mathsf{only}\ aw \vee [b]\mathsf{only}\ w, {\sim}b$. 143
6.6 A *broadcaster* and a detached channel. 144
6.7 An example of *hook* usage. 146
6.8 A coordination-based solution. 147
6.9 An *alternate merger*. 148

8.1 Pipes and Filter 166
8.2 Abstract Data Types and Objects 167
8.3 Layered Systems 168
8.4 Blackboard 169
8.5 Interpreter 170

Abstract

Modern computer based systems are essentially based on the cooperation of distributed, heterogeneous component organized into open software architectures that, moreover, can survive in loosely-coupled environments and be easily adapted to changing application requirements. Such is the case, for example, of applications designed to take advantage of the increased computational power provided by massively parallel systems or of the whole business of Internet-based software development.

In order to develop such systems in a systematic way, the focus in development method has switched, along the last decade, from functional to structural issues: both data and processes are encapsulated into software units which are connected into large systems resorting, to a number of techniques intended to support reusability and modifiability.

Actually, the complexity and ubiquity achieved by software in present times makes it imperative, more than ever, the availability of both technologies and sound methods to drive its development. Programming '*in–the–large*', *component–based* programming and *software architecture* become popular expressions which embody this sort of concerns and correspond to driving forces in current software engineering.

In such a context this thesis aims at introducing formal models for software connectors as well as the corresponding notions of equivalence and refinement upon which calculation principles for reasoning and transforming connector-based software architectures can be developed. This research adopts an exogenous coordination point of view in order to deal with components' temporal and spatial decoupling and, therefore, to provide support for looser levels of inter-component dependency.

The thesis also characterises a notion of *behavioural interface* for components

and services. Interfaces and connectors are put together to form *configurations*, an abstraction for representing software architectures.

A prototype implementation of a subset of the proposed models is provided, in the form of a Haskell library, as a proof of concept. Furthermore, the thesis reports on a case study in which exogenous coordination is applied to the specification of interactive systems.

Resumo

Um número crescente de sistemas computacionais é baseado na cooperação de componentes interdependentes e heterogêneas, organizadas em arquiteturas abertas capazes de sobreviverem em ambientes altamente distribuídos e facilmente adaptáveis a alterações nos requisitos das aplicações que os suportam. Tal é o caso, por exemplo, de aplicações que exploram o poder computacional de sistemas massivamente paralelos ou de sistemas desenvolvidos sobre a Internet.

Para desenvolver este tipo de sistemas de forma sistemática, o foco nos métodos de desenvolvimento alterou-se, ao longo da última década, dos aspectos funcionais para os aspectos estruturais dos sistemas: ambos, estruturas de dados e processos são encapsulados em unidades computacionais que são conectadas em grandes sistemas utilizando-se de diversas técnicas que se pretendem capazes de suportar a reutilização e a adaptabilidade do software.

Na realidade, a complexidade e ubiqüidade atingidas pelo software nos dias correntes tornam imperativo, mais do que nunca, a disponibilidade de tecnologias e sólidos métodos para conduzir este processo de desenvolvimento. Programação '*em-grande-escala*', programação *baseada em componentes* e *arquiteturas de software* são expressões populares que englobam esta preocupação e correspondem aos esforços direcionados pela engenharia de software.

Em tal contexto, esta tese tem por objetivo introduzir modelos formais para conectores de software bem como as correspondentes noções de equivalência e refinamento que suportem cálculos para raciocinar e transformar arquiteturas de software baseada em conectores. Esta pesquisa adota um ponto de vista de coordenação exógena para lidar com a separação espacial e temporal das componentes e suportar níveis elevados de independência entre componentes.

A tese caracteriza, ainda, uma noção de *interface comportamental* para componentes e serviços. Interfaces e conectores agregam-se para formar *configurações*, uma abstração introduzida para representar arquiteturas de software.

A implementação, em protótipo, de parte dos modelos propostos, sob a forma de uma biblioteca em HASKELL, é fornecida como prova de conceito. Finalmente, a tese percorre um estudo de caso em que coordenação exôgena é utilizada na especificação de sistemas interactivos.

Acknowledgements

First of all, I would like to thank FCT, the Portuguese Foundation for Science and Technology by the financial support of this research under a grant nr SFRH/BD/11083/2002.

From now on I will ask permission to the reader and shift the language from English to *Portuguese*.

Quero agradecer aos maiores incentivadores desta jornada, meus pais, Raul e Heloisa, pelo longo período de ausência física porém, nunca emocional. Vocês sempre estiveram comigo e foram um importante alicerce para este projeto. Agradeço também minhas irmãs Raquel, Luciana e Fabiana, também pelo apoio emocional e pela torcida para que tudo desse certo.

Agradeço ao meu grande amigo Giovani Rubert Librelotto, meu companheiro de outras jornadas acadêmicas e profissionais. Tu sabes da tua importância neste trabalho pois foi um dos grandes responsáveis em apresentar-me o professor Luís Barbosa e por ter dado todo o suporte quando cheguei em Portugal. Gigio, obrigado por tudo!

No período que morei em Portugal conheci muitos pessoas que ajudaram a minimizar a saudade e a distância do Brasil. Alfrânio, Gustavo, Ricardo, Fábio, Thiago e Victor. Graças a vocês, os 'brazucas_braga' a jornada foi ainda mais prazerosa.

Durante minha estada em Portugal juntaram-se a nós Ronnie e Ana. Sem dúvida outros amigos importantes para este processo. Ronnie foi um grande colega de gabinete e um grande companheiro nos cafés.

Tenho que agradecer também ao meu padrinho e amigo, Ogi Librelotto, pelo apoio em todos os momentos, bons e ruins, sempre disposto a ouvir e dar uma palavra de alento.

Em Portugal conheci grandes pessoas, amigos muito queridos. Primeiro quero agradecer ao David Costa, por sua amizade e acolhida a todos os brasileiros que chegavam ao Departamento de Informática da Uminho. Agradeço também pela ajuda ciêntifica, estava sempre disposto a auxiliar nas dúvidas que eu tinha sobre Haskell e outros assuntos.

Agradeço ao Pedro Gabriel, pelo amizade, pelo agradável convívio no gabinete, pelos cafés e pelos finos.

Agradeço também aos companheiros de grupo Nuno Rodrigues e Jacome Cunha. A ajuda de vocês foi também muito importante ao longo deste trabalho.

Agradeço a equipa técnico/administrativa do DI pelo suporte e a atenção que sempre dispendem aos alunos e professores. Vou citar o nome da Paula Anjo e do Jaime para que estes representem o meu agradecimento a todos.

Agradeço ao Grupo LMF pelo convívio, aprendizado e pelos saborosos 'Pure-Cafe's que "degustei" nestes anos de projeto.

Quero agradecer ao professor Pedro Henriques pela maneira tão acolhedora e bondosa de receber não apenas a mim, mas todos os brasileiros não apenas no DI, mas em sua própria casa.

Quero agradecer em especial minha namorada–noiva–esposa, Adriana. Ao longo desta jornada, desde o primeiro contacto com o professor Luís, ela estava ao meu lado. Mesmo sabendo que meu plano era ir viver ao longe ela apoiou me o tempo todo. A dor da saudade do primeiro ano em Portugal reverteu-se na grande alegria de ter você ao meu lado como esposa. Muito obrigado meu amor!

Minha estada em Portugal acabou por gerar mais que uma tese de doutorado gerou aquela que hoje é a grande razão do meu viver, minha querida filha Júlia.

Por fim, propositadamente deixo o meu orientador, não pela ordem de importância mas por esta ser a pessoa a quem mais devo agradecimentos. Devo agradecer, claro, por este trabalho que chega ao fim, mas não só, devo agradecer pela confiança depositada, pela acolhida que tive de sua parte, pelo apoio que sempre deu me para que eu pudesse crescer cientificamente. Agradeço pela amizade e atenção com que tratou a mim e minha esposa nestes anos. Afinal, no momento que vou defender esta tese minha esposa irá defender o mestrado que foi também possível por sua ajuda. Professor Luís Barbosa, falar de sua capacidade docente não é necessário por quem convive com o senhor a reconhece, quero exaltar sua capacidade humana, seu rigor científico misturado com sua generosidade o tornam um exemplo de professor, pesquisador e ser humano, professor Luís, ao senhor o meu muito obrigado por tudo!

Chapter 1

Introduction

Summary

This chapter provides the context for and states the problem addressed in this thesis: the development of formal models for software connectors and the study of their properties. The thesis structure and contributions are outlined, providing a 'roadmap' for the chapters to follow.

1.1 Motivation, objectives and research questions

Continuous evolution towards very large, heterogeneous, highly dynamic computing systems requires innovative approaches to master their complexity. Complex software systems are built by plugging components and services together which interact by exchanging data, performing computation, and modifying their environment. They are usually dynamic entities, running on different platforms, often owned by different organisations, interacting through public interfaces, and typically remaining loosely coupled, if not utterly unaware of each other.

Designing such systems right is very difficult, because their complexity is beyond the current practical reach of formal methods. Additional difficulties arise with third-party services, often under-specified or failing to meet their specifications.

Over the last decade component-based software development [Szy98, WW99] emerged as a promising paradigm to deal with the ever increasing need for mastering complexity in software design, evolution and reuse. As a paradigm it retains from object-orientation the basic principle of encapsulation of data and code, but shifts the emphasis from (class) inheritance to (object) composition to avoid interference between the former and encapsulation and, thus, paves the way to a development methodology based on *third-party assembly* of components. This is often illustrated by the visual metaphor of a *palette* of computational units, treated as black boxes, and a *canvas* into which they can be dropped. Connections are established by drawing *wires*, corresponding to some sort of interfacing code.

The expression *component-based programming*, although it has been around for a long time, became a buzzword in mid 1990's (see, e.g., [ND95, Szy98]). The basic motivation is to replace conventional programming by the composition and configuration of reusable off–the–shelf units, often regarded as *'abstractions with plugs'*. In this sense, a *component* is a 'black-box' entity which both provides and requires services, encapsulated through a public interface, which may exhibit both static and behavioural information.

In practice, however, software components do not fit together as *Lego* pieces. Moreover, as important as components themselves are the ways in which they can be put together to interact and cooperate in order to achieve some common goal. This motivated the development of new research questions concerning component adaptation, wrapping, composition and interaction. There is a number of answers to such questions, often formulated from disparate point of views, either technological, methodological or foundational. From the pragmatic, techonology-centred view, typically reducible to the famous Woody Allen's aphorism (*the answer is yes, but would you mind to repeat the question?*), to the most hard, if not exoteric, formal proposal, we have to recognize that component engineering is still in its infancy. Moreover, as it happened before with object-orientation, and software engineering in a broad sense, the paradigm has grown up to a collection of popular technologies, methods and tools, before consensual definitions and principles (let alone formal foundations) have been put forward.

There are essentially two ways of regarding, conceptually, *component-based* software development. The most wide-spread, which underlies popular technologies like, *e.g.*, Corba, DCom or JavaBeans, reflects what could be called the *object orientation* legacy. A component, in this sense, is essentially a collection of objects and, therefore, component interaction is achieved by mechanisms implementing the usual *method call* semantics. As F. Arbab stresses in [Arb03] this

> induces an asymmetric, unidirectional semantic dependency of users (of services) on providers (...) which subverts independence of components, contributes to the breaking of their encapsulation, and leads to a level of inter-dependence among components that is no looser than that among objects within a component.

An alternative point of view is inspired by research on coordination languages [GC92, PA98] and favours strict component decoupling in order to support a looser inter-component dependency. In this view, computation and coordination are clearly separated, communication becomes *anonymous* and component interconnection is externally controled. This model is (partially) implemented in JavaSpaces on top of Jini and fundamental to a number of approaches to component-based development which identify communication by generic channels as the basic interaction mechanism — see, *e.g.*, Reo [Arb03] or Piccola [NA03].

The research reported in this thesis affiliates itself in the latter point of view. Our aim was to investigate *formal models* for *gluing code* to orchestrate components of different origins and with different purposes.

In such a context, the concept of *software connector* emerged as a key one. The expression was coined by software architects to represent the interaction patterns among components, the latter regarded as basic computational elements or information repositories. Actually they are even mentioned in the pioneering papers on the then emerging Software Architecture discipline, in the early 1990's [PW92, GS93]. A *software connector* aims at mediating the communication and coordination activities among components, abstracting the actual gluing code between them. Examples range from simple channels or pipes, to event broadcasters, synchronisation barriers or even more complex structures encoding client-server protocols or hubs between databases and applications.

Therefore, the problem to be addressed in this PhD project became formulated in terms of the following research questions:

- What is a software connector and how can it be formally characterised in order to became an useful abstraction to reason about software design in-the-large?
- In which ways can software connectors be composed?
- Is there an algebra of software connectors upon which to formulate and discuss equivalence and refinement of connector-based designs?

These research questions lead to the development of an hierarchy of three models of software connectors, progressively enriched, which constitutes the central contributions of the thesis. In all cases, an algebra was defined and part of the underlying structure studied. The mentioned hierarchy goes from purely synchronous, stateless connectors, to stateful ones and, finally, to connectors which are context-aware in the sense of exhibiting context-dependent behaviour.

The specific contributions of the thesis are enumerated in section 1.2. Afterwards, section 1.3 covers, in some detail, its research context. Finally, a few mathematical notions, on functions, relations and coalgebras, are briefly reviewed in section 1.4 for future reference.

We should not, however, end this section without making two comments which we believe help to situate this work in its most direct influences.

A major influence of this work was the previous body of research at Minho Formal Methods group on coalgebraic models for components and services, documented in [Bar00, BO03, Bar03, CBO05, MB05, BO06]. Initially this thesis was thought as a follow-up of such work. Soon, however, our focus shifted from *components* to *connectors*: instead of an algebra of components, this work centred in an algebra of connectors, more appropriated, from our point of view, to deal correctly with exogenous coordination problems.

In the meantime, *i.e.*, around 2003-2004 where our work began, the Reo framework was emerging and attracting a lot of research efforts and resources. Most of our aims were also shared by the Reo community. In a sense this fact reshaped our own work: what is presented here as the thesis outcome can very well be seen as a specific contribution to the semantics of Reo-like frameworks. A distinguished feature of the work reported here with respect to mainstream literature on Reo, is our insistence on the explicit definition of combinators for composing connectors, as a more algebraic alternative to composition based on graph transformation. On the other hand, the quest for a semantics for connectors able to propagate context constraints, which was for a long time a main open problem in Reo, was set of a research objective for our work. Our own solution, detailed in chapter 5, was presented, even if in a partial formulation, at Foclasa'06, in Bonn, and later published in [BB09]. In any case the influence of Reo and the effective contact with the Reo group at CWI, Amsterdam, cannot be underestimated in the outcome of this thesis.

1.2 Thesis structure and contributions

The thesis introduces three formal models for *software connectors*, each one endowed with a set of combinators and notions of equivalence and refinement. Connectors are intended to coordinate components or services and, therefore, the thesis also discusses the specification of behaviour in *component interfaces*. Finally, component interfaces and connectors are put together in what we called a *configuration*, which amounts to an abstraction of a software architecture, and their joint behaviour computed.

Most of the results presented were previously published in a series of scientific publications: 3 journal papers, 2 conference papers and 3 papers in refereed workshops. In the sequel we sum up each chapter contribution and its association to published papers.

Chapter 2 is based on

> M. A. Barbosa and L. S. Barbosa. *A relational model for component interconnection*. Journal of Universal Computer Science, 10(7):808-823, 2004.

It introduces a relational model for software connectors which do not exhibit any form of internal memory (or local state). Therefore, a connector is modeled as a relation between values present at its input and output ports. Of course, only simultaneous observation of ports is allowed. Despite this restriction in expressiveness, the model is both simple and intuitive, and useful in a number of relevant applications. Connector equivalence and refinement correspond to relational equality and inclusion, respectively.

Chapter 3 introduces a different model, resorting to coalgebras to model software connectors. The chapter is based in

> M. A. Barbosa and L. S. Barbosa. *Specifying software connectors*. In K. Araki and Z. Liu, editors, 1st International Colloquium on Theorectical Aspects of Computing (ICTAC'04), pages 53-68, Guiyang, China. Springer Lect. Notes Comp. Sci. (3407). 2004.

The model is intended to capture software connectors whose input-output behaviour is partially determined by a memory of past computations encoded as the connector's state space. Therefore each connector becomes a coalgebra for a functor capturing its signature of communication ports. Standard coalgebraic techniques apply; in particular connector equivalence boils down to bisimulation. In brief this models extends the one presented in chapter 2 in exactly the same sense that transition systems extend binary relations: coalgebras are just *relations extended in time*.

The chapter also discusses how some form of mobility can be expressed within the model. It introduces a special connector, the *orchestrator*, whose role is to manage interconnection patterns that can change at run-time. A limitation of this solution is its too operational character. This contribution was presented in

> M. A. Barbosa and L. S. Barbosa. *An orchestrator for dynamic interconnection of software components*. In Proc. 2nd International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord'06), volume 181 of Electronic Notes in Theoretical Computer Science, pages 49-6. Elsevier, 2007.

Chapter 4 suspends the introduction of connector models, to shift attention to component interfaces. Actually, making software connectors *context-aware*, *i.e.*, able to sense their environment and actuate accordingly, entails a deeper update of our basic model. In particular it becomes necessary to endow interfaces, of both components and connectors, with a specification of their behaviour or interaction protocol. Process algebras are a natural candidate for such specifications. This, however, entails the need for more generic and adaptable approaches to their design. For example, similar combinators coexisting with different interaction disciplines may be required. In such a context, chapter 4 pursues a research programme on a coinductive rephrasal of classic process algebra documented in [Bar01, BO02]. A particular emphasis is put on the study of interruption combinators defined by natural co-recursion. The chapter also illustrates the verification of their properties in an pointfree reasoning style as well as their direct encoding in Haskell. Finally, behaviour-annotated interfaces are defined paving the way to the third connector model to be presented in chapter 5. The idea of behavioural interfaces, in the context of this research, was first introduced in

> M. A. Barbosa, L. S. Barbosa, and J. C. Campos. *Towards a coordination model for interactive systems*. In A. Cerone and P. Curzon, editors, FMIS 2007: Proc. 1st Inter. Workshop in Formal Methods for Interactive Systems, volume 347 of Electronic Notes in Theoretical Computer Science, pages 89-103. Elsevier, 2007.

The technical contributions of the chapter were published in

> P. Ribeiro, M. A. Barbosa, and L. S. Barbosa. *Generic process algebra: A programming challenge*. Journal of Universal Computer Science, 12(7):922-937, 2006.

Chapter 5 is devoted to the third of the three models for software connectors discussed in the thesis. It builds on the behavioural interfaces discussed in chapter 4 and is based on

> M. A. Barbosa and L. S. Barbosa. *A perspective on service orchestration.* Science of Computer Programming, 74(9):671-687, 2009.

as a consolidation of some ideas first discussed in

> M. A. Barbosa and L. S. Barbosa. *Configurations of web services.* In FOCLASA'06: Proc. 5th Inter. Workshop on the Foundations of Coordination Languages and Software Architectures, volume 175 (2) of Electronic Notes in Theoretical Computer Science, pages 39-57. Elsevier, 2006.

The model was designed to deal correctly with context dependent behaviour and its propagation. Actually, capturing context dependent behaviour and ensuring it is suitably propagated through complex connector networks, is a difficult problem, still not addressed, for example, in the most popular semantics for Reo. A second contribution of the paper is a notion of *configuration*, which abstracts away fragments of a software architecture, putting together connectors and components' interfaces.

Chapter 6 was originally thought as a case-study, but it turned out to be a little more than that, introducing a logic for specifying connectors. It is based on

> Marco A. Barbosa, L. S. Barbosa, and José C. Campos. *A coordination model for interactive components.* In F. Arbab and M. Sirjani, editors, Proc. of FSEN 2009, Kish, Iran. Springer Lect. Notes Comp. Sci. (to appear), 2009.

The case study was about *interactors*, thought as abstract characterisations of interactive components. Interactors are well-known in the area of formal modelling techniques for interactive systems. The chapter proposes to replace traditional, hierarchical, 'tree-like' composition of interactors in the specification of complex interactive systems, by their *exogenous coordination* through general-purpose software *connectors* which assure the flow of data and the meet of synchronisation constraints. The chapter's technical contribution is twofold. First a modal logic is defined to express behavioural properties of both interactors and connectors. The logic is new in the sense that its modalities are indexed by fragments of *sets* of actions to cater for action co-occurrence. Then, this logic is used in the specification of both interactors and coordination layers which orchestrate their interconnection, providing a case-study in the use of software connectors for program coordination.

Chapter 7 introduces a set of basic programming primitives which implement some of the main concepts and models described in the thesis, for prototyping purposes. Components' interfaces and connectors are encoded in Haskell. The former are defined by constructing generic ports allowing anonymous communication among components. Although not expected to be a proof-of-concept for the models introduced in the thesis, this library allows the software architect to ’play’ with connectors and configurations when designing a new system.

Chapter 8, finally, concludes the thesis and enumerates a number of issues for future work. In particular, it is suggested that the connector models proposed in the thesis may of use to formalise software architectural patterns and styles, which justifies a somewhat detailed review of this area as a landscape for future research.

1.3 Context: Components, coordination and architectures

The problem addressed in this thesis can be framed into three main research areas emerging in Software Engineering: *software components*, *software architecture* and *coordination models and languages*. This section provides a brief overview of such a context.

1.3.1 Components

The 1990s have shown that object-oriented technology alone is not enough to cope with the rapidly changing requirements of present day applications. One of the reasons is that, although object-oriented methods encourage one to develop rather expressive models that reflect the objects of the problem domain, this does not necessarily yield software architectures that can be easily adapted to changing requirements. In particular, object–oriented methods do not typically lead to designs that make a clear separation between computational and compositional aspects; this separation is, however, a hallmark of component-based systems [SN99].

Actually, component-based software development offers a plausible solution to one of the toughest and most persistent problems in software engineering: how to effectively maintain software systems in the face of changing and evolving requirements. Software systems, instead of being programmed in the conventional sense, are constructed and configured using libraries of components. Applications can be adapted to changing requirements by reconfiguring components, adapting existing components, or introducing new ones. Component-based systems, achieve flexibility by clearly separating the stable of the system (components) from the specification of their composition.

Components are black-box entities that encapsulate services behind well-defined interfaces. These interfaces tend to be very restricted in nature, reflecting a particular model of plug-compatibility supported by a component framework, rather than being very rich and reflecting real world entities of the application domain. Note, however, that components are not used in isolation, but according to a *software architecture* [SG96] that determines the interfaces that components may have and the rules governing their composition.

Actually, a software *component* provides and requires services. These services can be seen as *plugs* (or, more prosaically, interfaces). The added value of components comes from the fact that the plugs must be standardized (i.e. a component must be designed to be composed) [ND95]. A component that is not plug-compatible with anything can hardly be called a component. The plugs of a components take many different shapes, depending on whether the component is a function, a template, a class, a data-flow filter, a widget, an application, or a server. It is important to note that components also require services, as this makes them individually configurable (e.g., consider a sorting component that behaves differently given different containers or comparison operators [MS96]).

Still, it may be necessary to adapt the behaviour of components in order to compose them. Such adaptation is needed whenever components have to be used into systems that they have not been designed for. Adaptations of this kind, however, are often nontrivial (if not impossible) and considerable glue code may be needed to reuse components coming from different frameworks [GAO95]. In general glue code is rather application specific and cannot to be reused in different settings, unless well-understood glue abstractions can be used.

Component-based software development is always driven by an underlying component framework. A component framework offers a predefined set of reusable and plug-compatible components and defines a set of rules specifying how components can be instantiated, adapted, and composed. Component-based software development has the advantage that applications do not have to be developed from scratch: new systems can benefit from well-understood properties and important design decisions of previous systems, leading to increased flexibility and adaptability during maintenance and evolution.

Component-based applications provide added value over conventionally developed applications, since they are easier to adapt to new and/or changing requirements. This is the case since one can:

- configure and adapt individual components;
- unplug components and plug in others;
- reconfigure the connections between sets of components at a high level of abstractions;

- define new plug-compatible components from either existing components or from scratch;
- take legacy components and adapt them to make them plug-compatible, and
- treat a composition of components as a component itself.

In programming practice, components come often associated to some form of *scripting languages* [NTdMS91]. Whereas conventional programming languages are perfectly suitable for implementing software components, scripting languages are designed for configuring and connecting components. The use of scripting languages encourages the development of reusable components highly focused on the solution of particular problems, and the assembly of these components by means of scripts. A *script* specifies how components are plugged together. It may be seen like the script that tells actors how to play various roles in a theatrical piece. The essence of a true scripting language will also let we treat a script as a component: a Unix shell script, for example, can be used as a Unix command within other scripts. Actually, a script makes architectures explicit by exposing exactly how the components are connected.

1.3.2 Architectures

Components are by definition elements of a *component framework* or *architecture*: they adhere to a particular *architectural style* that defines the plugs, the connectors, and the corresponding composition rules. A *component framework* is a collection of software components and architectural styles that determines the interfaces that components may have and the rules governing their composition. In contrast to an object-oriented framework where an application is generally by subclassing framework classes that respond to specific application requirements, a component framework primarily focuses on object and class (i.e., component) composition. A *connector* is the wiring mechanism used to plug components together [SG96]. Again, depending on the kind of components they deal with, connectors may or may not be present at run-time: contrast, for example, C++ template compositional to Unix pipes and filters.

Software architecture emerged as a proper discipline[PW92, GS93, AG97, Gar03, BCK03] in Software Engineering, from the need to explicitly consider, in the development of increasingly bigger and more complex systems the effects, problems and opportunities of the system's overall structure, organisation and emergent behaviour. In a broad definition, the architecture of a system describes its fundamental organisation, which illuminates the top level design decisions, namely

- how is it composed and of which interacting parts?
- which are the interactions and communication patterns present?
- which are the key properties of parts the overall system rely and/or enforce?

As a model it acts as an abstraction of a system that surpresses details of elements that do not affect how they use, are used by, relate to or interact with other elements. Therefore it focus on the structural elements and their interfaces by which a system is composed, their separate and joint behaviour as specified in collaborations among those elements, and finally the composition of these structural and behavioral elements into larger subsystems. According to norm ANSI/IEEE Std 1471-2000, which is part of a on-going standardisation effort, it describes *the fundamental organisation of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.*

The primary focus of software architectures is the identification of components that are necessary for the architecture, design, and implementation of a system. The importance of software architectures in the software life-cycle is often understimated. Having reusable *architectural styles* is a precondition of successfully developing reusable components and frameworks [BCK03] and, therefore, for building component-based applications.

Architectural description languages (ADL) are intended to specify and reason about architectural styles [SG96]. An ADL is a notation that allows for a precise description of the externally visible properties of a software architecture, supporting different architectural styles at different levels of abstraction. Externally visible properties refer to those assumptions other components can make of a component, such as its provided services, performance characteristics, error handling, and shared resource usage.

More will be said, by the end of this thesis, on *architectural styles*, as a challenge for extended application of some of our results. For the moment, however, consider, as an illustration, a very elementary example of an architectural design, well-known of all programmers. The example is the following UNIX shell script which reverses the lines of a 7-bit character input stream:

```
cat -n | sort -r -n | cut -b8-
```

Analyzing this shell script, we can immediately identify components and connectors as well as the underlying architecture: the script consists of:

1. a data source (i.e., the standard input stream of cat);
2. three components (the UNIX processes cat, sort, and cut;
3. two connectors (i.e character streams); and
4. a data sink (the standard output of cut).

The components and the character streams of the script form a pipeline, where each component only depends on the output of its predecessor. Since all shell scripts fulfill similar restrictions, they all share a similar overall structure. They conform to a *pipe and filter* architectural style [SG96].

1.3.3 Coordination

A new class of formalisms has recently evolved for describing concurrent and distributed computations based on the concept of *coordination*. In a sense, coordination can be considered as the scripting of concurrent and distributed components.

the coordination paradigm [JMA96, Arb98], claiming for a strict separation between effective computation and its control, appeared as a solution to the problem of managing interaction among concurrent activities in a system. emerged from the need to exploit the full potential of massively parallel systems which requires models able to deal, in an explicit way, with the concurrency of cooperation among very large number of heterogeneous, autonomous and loosely-coupled components. Coordination models [GC92, PA98, Arb03] make a clear distinction between such components and their interactions and focus on their joint *emergent* behaviour.

Traditionally, coordination models and languages have evolved around the notion of a shared dataspace — a memory abstraction accessible, for data exchanging, to all processes cooperating towards the achievement of a common goal. The first coordination language to introduce such a notion was Linda [ACG86]; many related models evolved later around similar notions [JMA96, BCG97]. The underlying model was *data-driven*, in the sense that processes can actually examine the nature of the exchanged data and act accordingly.

An alternative family of models, called *event-driven* or *control-driven*, more suitable to systems whose components interact with each other by posting and receiving events, the presence of which triggers some activity. A pioneer model in this family is Manifold [AHS93], which implements the Iwim model [Arb96]. Contrary to the case of the data-driven family where coordinators directly handle data values, in these models processes are regarded as black boxes and communicate with their environment by means of clearly defined interfaces (often referred to as input or output ports).

Let us briefly review some popular coordination paradigms, within the family of *control-driven* models. The choice is justified by the emphasis placed in this thesis: as mentioned above, this research affiliates itself in overall approach of which Reo is the landmark representative.

- **Piccola** [ALSN01] is a language which applies the paradigm "Applications = Components + Scripts". Piccola models components and compositional abstractions by means of communicating concurrent agents. Flexibility, extensibility, and robustness are obtained by modeling both interfaces of components and the contexts they live in by "forms", a special notion of extensible records. The main language element in Piccola is a so-called *form expressions* that represents a unified concept of both π–agents and π-forms. In fact, forms expressions are sequences of form terms (e.g, synchronous and asynchronous functions calls, binding extensions). Using form expressions, Piccola programs or scripts can be defined without using that low-level primitives of the underlying in π-calculus. The parallel composition operator, for example, is modeled by asynchronous function calls whereas the rendezvous of input – and output – prefixes is achieved by a synchronous function call. Piccola has a syntax similar to that of Python and Haskell.

- **Iwim** [Arb96] is a communication model that avoids the shortcomings of a typical message passing model and can be used in a paradigm to construct complex cooperation protocols. The basic concepts in the Iwim model are *processes*, *events*, *ports*, and *channels*. A process is a *black box* with well defined connection ports through which it exchanges *units* of information with the other processes in the environment. A port is a named opening in the bounding walls of a process through which units of information are exchanged using standard I/O type primitives analogous to read and write. Without loose of generality, we assume that each port is used for exchange of information in only one direction: either into (input port) or onto of (output port) a process. It is used the notation $p.i$ to refer to the port i of the process instance p. The interconnections between the ports of processes are made through channels. A channel connects a (port of a) producer (process) to a (port of a) consumer (process). The notation $p.i \rightarrow q.i$ denotes a channel connecting the port o of the producer process p to the port i of the consumer process q.

- **Manifold** [AHS93] is a coordination language based on the Iwim model. It is a language whose sole purpose is to manage complex interconnections among independent, concurrent processes. The basic components in the MANIFOLD model of computation are *processes*, *events*, *ports*, and *streams*. A process is a *black box* with level well defined connection ports through which it exchanges *units* of information with the other processes int its environment. The internal operation of some of these black boxes are indeed written int the MANIFOLD language, which makes it possible to open them up, and describe their internal behavior using the MANIFOLD model. These processes are called *manifolds*. Other processes may in reality be pieces of hardware, programs written in other programming languages, or human beings. These processes are called *atomic processes* in MANIFOLD.

- **REO** ($P\epsilon\omega$) [Arb03, Arb04] is a paradigm for composition of software components based on the notion of mobile channels. It is a channel-based exogenous coordination model wherein complex coordinators, called *connectors*, are compositionally built out of simples ones. The simplest connectors in Reo are a set of *channels* with well-defined behavior supplied by users. REO can be used as a language for coordination of concurrent processes, or as a "glue language" for compositional construction of connectors that orchestrate component instances in a component-based systems. The emphasis in REO is on connectors and their composition only, not on the entities that connect to, communicate and cooperate through these connectors. Each connector in REO imposes a specific coordination pattern on the entities (e.g., components) that perform I/O operations through that connector, without assuming any the knowledge of these entities.

The purpose of a coordination model is similar to that of software architectures: making a clear separation between computational elements and their relationships by proving abstractions for controlling synchronization, communication, creation and termination of concurrent and distributed computational activities [Arb96] . One can also consider coordination as the scripting of concurrent and distributed components.

For the last 15 years, the emergence of massive concurrent, heterogeneous systems and the growing complexity of interaction protocols has brought coordination to a central place in software development. Such development contributed to broadening its scope of application and entailed the development of a number of specific models, languages and semantics. Coordination languages have been applied in several contexts. For example, to the parallelization of computation intensive sequential programs in the fields of simulation of Fluid Dynamics systems, matching of DNA strings, molecular synthesis, parallel and distributed simulation, monitoring of medical data, computer graphics, analysis of financial data integrated into decision support systems, and game playing (chess). See ([AHDLM96], [CH96], [GM97]) for some concrete examples.

1.4 Mathematical preliminaries

This section reviews a number of mathematical concepts and notation used in the thesis. It is not exhaustive and can be easily skipped on first reading. The section is organised into three parts: on *pointfree functional notation*, the *calculus of binary relations* and *coalgebra*. Reference [BM97] is an excellent textbook for the categorical-inspired calculi of both functions and relations. An alternative reference for the latter is [BH93]. Note that, due to its ability to represent partiality and nondeterminacy, the relational calculus, specially in its pointfree style, has been extensively used in Computer Science. Reference [Fok92a] is mandatory reading on the envisaged calculational style.

Several phenomena in computing are hardly definable (or even simply not definable) as algebras, *i.e.*, in terms of a complete set of constructors. For example, processes, transition systems, objects, stream-like structures used in lazy programming languages, 'infinite' or non well-founded objects arising in semantics, and so on. Such 'systems' are inherently dynamic, do possess an observable behaviour, but their internal configurations remain hidden and have therefore to be identified if not distinguishable by observation. Furthermore, the formalisation of the possibly infinite behaviour they may exhibit requires infinite structures, whereas elements of an initial algebra are always finite. *Coalgebraic structures* seem appropriate to deal with such issues. References [Rut00] and [Jac02] are excellent introductions to the topic.

From the point of view of a Computer scientist, it is remarkable that the basic properties captured in the categorical framework [Mac71, AHS90, McL92] — such as *universality*, *functoriality* and *naturality* — can be phrased in a 'calculational' style. This means that such properties can be formulated as (usually equational) laws and used to manipulate and reason about objects and arrows of the underlying category. Such a 'calculational' style matches nicely a main concern in computer science — the seek for program calculi able to promote programming to a modern engineering discipline.

1.4.1 Functions

In the sequel, as an illustration of the pointfree functional calculus, the *product* and *coproduct* constructions are recalled and some associated laws that turn out to be most useful in calculation. Product and coproduct are the categorical generalisation of Cartesian product and disjoint union in an universe of sets. In a sense, they capture the duality between *co-occurrence* and *choice*, which may explain their major role in modelling computational systems.

Products & Splits. Functions with a common domain can be glued through a *split* $\langle f, g\rangle$ as shown in the following diagram:

$$\begin{tikzcd}
 & Z \arrow[ld, "f"'] \arrow[d, "\langle f,g\rangle"] \arrow[rd, "g"] & \\
A & A \times B \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & B
\end{tikzcd}$$

which defines the product of two sets by the following universal property:

$$k = \langle f, g\rangle \quad \equiv \quad \pi_1 \cdot k = f \ \wedge \ \pi_2 \cdot k = g \tag{1.1}$$

from which the following properties can be derived:

$$\langle \pi_1, \pi_2\rangle \ = \ \mathsf{id}_{A\times B} \tag{1.2}$$

$$\pi_1 \cdot \langle f, g\rangle = f \ , \pi_2 \cdot \langle f, g\rangle = g \tag{1.3}$$

$$\langle g, h\rangle \cdot f \ = \ \langle g \cdot f, h \cdot f\rangle \tag{1.4}$$

$$(i \times j) \cdot \langle g, h\rangle \ = \ \langle i \cdot g, j \cdot h\rangle \tag{1.5}$$

known respectively as × *reflection*, *cancelation*, *fusion* and *absorption* laws. Similarly arises *structural equality*:

$$\langle f, g\rangle = \langle k, h\rangle \ \equiv \ f = k \ \wedge \ g = h \tag{1.6}$$

Finally note that the product construction is *functorial*:

$$f \times g = \lambda \langle a, b\rangle \ . \ \langle f\ a, g\ b\rangle \tag{1.7}$$

Sums & Eithers. Dually, functions sharing the same codomain may be glued together through an *either* combinator, expressing alternative behaviours, and introduced as the universal arrow in a datatype sum construction. $A + B$ is defined as the target of two arrows $\iota_1 : A \longrightarrow A + B$ and $\iota_2 : B \longrightarrow A + B$, called the *injections*, which satisfy the following universal property: for any other set Z and functions $f : A \longrightarrow Z$ and $g : B \longrightarrow Z$, there is a unique arrow $[f, g] : A + B \longrightarrow Z$, usually called the *either* (or *case*) of f and g, that makes the following diagram to commute:

$$\begin{tikzcd}
A \arrow[r, "\iota_1"] \arrow[dr, "f"'] & A+B \arrow[d, "{[f,g]}"] & B \arrow[l, "\iota_2"'] \arrow[dl, "g"] \\
 & Z &
\end{tikzcd}$$

Again this universal property can be written as

$$k = [f, g] \quad \equiv \quad k \cdot \iota_1 = f \ \wedge \ k \cdot \iota_2 = g \tag{1.8}$$

from which one infers correspondent *cancelation*, *reflection* and *fusion* results:

$$[f, g] \cdot \iota_1 = f \ , \ [f, g] \cdot \iota_2 = g \tag{1.9}$$

$$[\iota_1, \iota_2] \ = \ \mathsf{id}_{X+Y} \tag{1.10}$$

$$f \cdot [g, h] \ = \ [f \cdot g, f \cdot h] \tag{1.11}$$

Products and sums interact through the following *exchange* law

$$[\langle f, g\rangle, \langle f', g'\rangle] \ = \ \langle [f, f'], [g, g'] \rangle \tag{1.12}$$

provable by either product (1.1) or sum (1.8) universality. The *sum* combinator also applies to functions yielding

$$f + g : A + B \longrightarrow A' + B' \tag{1.13}$$

defined as $[\iota_1 \cdot f, \iota_2 \cdot g]$.

Conditional expressions are modelled by coproducts. In this paper we adopt the McCarthy conditional constructor written as $(p \ \rightarrow \ f,\ g)$, where $p : A \longrightarrow \mathbf{2}$ is a predicate. Intuitively, $(p \ \rightarrow \ f,\ g)$ reduces to f if p evaluates to true and to g otherwise. The conditional construct is defined as

$$(p \ \rightarrow \ f,\ g) \ = \ [f, g] \cdot p?$$

where $p? : A \longrightarrow A + A$ is determined by predicate p as follows

$$p? = \qquad A \xrightarrow{\langle \mathsf{id}, p\rangle} A \times (\mathbf{1} + \mathbf{1}) \xrightarrow{\mathsf{dl}} A \times \mathbf{1} + A \times \mathbf{1} \xrightarrow{\pi_1 + \pi_1} A + A$$

where dl is the distributivity isomorphism. The following laws are usefull to calculate with conditionals [Gib97].

$$h \cdot (p \ \rightarrow \ f,\ g) \ = \ (p \ \rightarrow \ h \cdot f,\ h \cdot g) \tag{1.14}$$

$$(p \ \rightarrow \ f,\ g) \cdot h \ = \ (p \cdot h \ \rightarrow \ f \cdot h,\ g \cdot h) \tag{1.15}$$

$$(p \ \rightarrow \ f,\ g) \ = \ (p \ \rightarrow \ (p \ \rightarrow \ f,\ g), (p \ \rightarrow \ f,\ g)) \tag{1.16}$$

1.4.2 Relations

Basics. Let $R : B \longleftarrow A$ denote a binary relation on (source) type A and (target) type B, and bRa stand for the representation of $\langle b, a\rangle \in R$. The set of relations from A to B is *ordered* by inclusion $\subseteq$, with relation equality being established by anti-symmetry. Fact $R \subseteq S$ means that relation S is either more defined or less deterministic than R, that is, for all a and b of the appropriate types, $bRa \rightarrow bS\,a$.

The algebra of relations is built on top of three basic operators: composition $(R{\cdot}S)$, meet $(R{\cap}S)$ and converse (R°). As expected, $aR^{\circ}b$ iff bRa, meet corresponds to set-theorectical intersection and $\cdot$ generalizes functional composition: $b(R \cdot S)c$ holds iff there exists some $a \in A$ such that $bRa \wedge aS\,c$.

Any function f can be seen as the relation given by its graph, which, in this paper, is also denoted by f. Therefore $bfa \;\equiv\; b = fa$. In this setting functions enjoy a number of properties of which the following is singled out by its role in the pointwise to pointfree conversion:

$$b\,(f^{\circ} \cdot R \cdot g)\,a \;\equiv\; (fb)\,R\,(ga) \tag{1.17}$$

Conversely, any relation $R : B \longleftarrow A$ can be uniquely transposed into a set-valued function $\Lambda R : \mathcal{P}B \longleftarrow A$, where the transpose operator Λ satisfies the following universal property:

$$f = \Lambda R \;\equiv\; (bRa \;\equiv\; b \in (fa)) \tag{1.18}$$

The interplay between functions and relations is also captured by the so-called *shunting* laws [BH93], of which the following is an example:

$$f \cdot R \subseteq S \cdot g \;\equiv\; R \subseteq f^{\circ} \cdot S \cdot g \tag{1.19}$$

Special relations on $A \times B$ include the bottom and top relations:

$$\bot \subseteq R \subseteq \top$$

where $B \xleftarrow{\top} A$ is the largest relation of its type (*universal relation*) and $B \xleftarrow{\bot} A$ is the smallest relation of this type (*empty relation*).

An order (or endo–relation) $A \xleftarrow{R} A$ is:

- reflexive: iff $id_A \subseteq R$

- coreflexive: iff $R \subseteq id_A$

- transitive: iff $R \cdot R^\circ \subseteq id_A$

- anty-symmetric: iff $R \subseteq R^\circ (\equiv R = R^\circ)$

- connected: iff $R \cup R^\circ = \top$

A relation is an *equivalence relation* if it is reflexive, symmetric and transitive and a relation is a *preorder* if it is transitive and reflexive. The combination of a set S and a binary relation on S that is a preorder is called a *preordered set*. When a relation is transitive, reflexive and anti-symmetric it is called a *partial ordering*. The combination of the set S and a partial ordering on the set is called a *partially ordered* set, or a *poset* for short. A relation R is a *total ordering* if it is a partial ordering and, fora all x and y, either xRy and yRx.

Many useful properties of relations have simple relation–algebraic formulations. In particular, a relation $B \xleftarrow{R} A$ is called *entire* (or *total*) if:

$$R \; is \; entire \equiv id_A \subseteq R^\circ \cdot R \tag{1.20}$$

and is called *simple* (or *functional*) if:

$$R \; is \; simple \equiv R \cdot R^\circ \subseteq id_B \tag{1.21}$$

Relation $R^\circ \cdot R$ is called the *kernel* of R; similarly, $R \cdot R^\circ$ is called its *image*:

$$\begin{aligned} \mathsf{ker}\, R &\triangleq R^\circ \cdot R \\ \mathsf{img}\, R &\triangleq R \cdot R^\circ \end{aligned}$$

Therefore,

$$R \; is \; entire \;\equiv\; id_A \;\subseteq\; \mathsf{ker}\, R \tag{1.22}$$

$$R \; is \; simple \;\equiv\; \mathsf{img}\, R \;\subseteq\; id_B \tag{1.23}$$

Simple relations are also called *partial functions*. Relations may also be surjective or injective:

$$R \text{ is surjective iff } R^\circ \text{ is entire} \tag{1.24}$$

$$R \text{ is injective iff } R^\circ \text{ is simple} \tag{1.25}$$

As usual, a relation R is bijective iff it is injective and surjective. Moreover,

$$R \text{ is entire and injective } \equiv \mathsf{ker}\, R = id \tag{1.26}$$

$$R \text{ is simple and surjective } \equiv \mathsf{img}\, R = id \tag{1.27}$$

The domain of a relation R is the set denoted by $\mathsf{dom}\ R$ and the range of a relation R is the set denoted by $\mathsf{rng}\ R$. Formally,

$$\mathsf{dom}\, R \;=\; id_A \cap \mathsf{ker}\, R \tag{1.28}$$

$$\mathsf{rng}\, R \;=\; id_B \cap \mathsf{img}\, R \tag{1.29}$$

Other properties include,
Identity:

$$R \cdot id \;=\; R \;=\; id \cdot R \tag{1.30}$$

Empty relation:

$$R \cdot \bot \;=\; \bot \;=\; \bot \cdot R \tag{1.31}$$

Universal relations:

$$R \cdot \top \;=\; \top \;=\; \top \cdot R \tag{1.32}$$

Associativity:

$$R \cdot (S \cdot T) \;=\; (R \cdot S) \cdot T \tag{1.33}$$

The category of sets and binary relations is denoted by Rel. References [BM97] and, mainly, [BH93], provide a detailed account of the calculus of binary relations, in a *pointfree* calculational style.

Galois connections. Several constructions in the relational calculus emerge as *Galois connections* [Bac02], $C \dashv C'$. Such is the case, for example, of the left and right *division* operators given, respectively by

$$\bullet R \dashv /R \quad \text{and} \quad R\bullet \dashv R\backslash$$

A lot of properties about relations are related with Galois connections. A Galois connection involves two preordered sets $(\mathcal{A}, \leq)$ and $(\mathcal{B}, \leq)$ and two functions, $F \in \mathcal{A} \leftarrow \mathcal{B}$ and $G \in \mathcal{B} \leftarrow \mathcal{A}$. These four components together form a Galois connection iff for all $x \in \mathcal{B}$ and $y \in \mathcal{A}$ the following holds

$$F \bullet x \leq y \equiv x \leq G \bullet y \tag{1.34}$$

The theory of Galois connections can entirely be developed for preordered sets, although it is occasionally simpler to assume a partial ordering.

1.4.3 Coalgebras

Coalgebras and infinite behaviours. In the semantics of programming, finite data types such as finite lists, have traditionally been modelled by initial algebras. Later the dual structure of final *coalgebras* [Rut00, Jac02] were introduced to deal with *infinite* data types, transition systems, automata, and several phenomena in computing which are hardly definable (or even simply not definable) as algebras, *i.e.*, in terms of a complete set of constructors.

Although previously known in Universal Algebra, coalgebras began to be seriously considered only after the categorical account of both algebraic and coalgebraic structures of a *type* T (*i.e.*, for an endofunctor T in an arbitrary category) has provided the right generic framework in which several phenomena and theories fit. The systematic study of their theory, essentially along the lines of Universal Algebra, was initiated by J. Rutten in [Rut00]. Reference [JR97] provides an introductory tutorial. The relevance of coalgebraic concepts and tools was first recognised in programming semantics — see, for example, P. Aczel foundational work on 'non well-founded sets' and the semantics of processes [Acz88, Acz93]; H. Reichel characterisation of behavioural satisfaction [Rei81] and J. Rutten, G. Plotkin and D. Turi work on final semantics [RT94, Tur96, TP97].

The notion of coalgebras is general enough to cover many types of systems and is specific enough to allow for quite a number of interesting results. They are typically used to describe state-based dynamical systems, where the state space (set of states) of the system is considered as a black box, and nothing is known about the way the observable behaviour is realised. Such 'systems' are inherently dynamic, do possess an observable behaviour, but their internal configurations remain hidden and have therefore to be identified if not distinguishable by observation.

While data entities in an algebra are built by constructors and considered to be different if differently constructed, coalgebras deal with entities which are observed, or decomposed, by observers (or 'destructors'). Any two internal configurations are identified if they cannot be distinguished by observation. Given an endofunctor F, a F-coalgebra is a map $p : U \longrightarrow F\ U$ which may be thought of as a transition structure, of *shape* F, on object U, usually referred to as its *carrier* or *state space*. The shape of F describes not only the way the state is (partially) accessed, through *observers*, but also how it evolves, through *actions*. F specifies a signature of actions and observers over a carrier but it omits its constructors. As a consequence equality has to be replaced by *bisimilarity* (*i.e.*, equality with respect to the observation structure provided by F) and *coinduction* replaces induction as a proof principle.

The dual concept to *initial* algebra is that of *final* coalgebra. For a given F, it consists of all possible behaviours up to bisimilarity, in the same sense that an initial algebra collects all terms up to isomorphism. It is also a (greatest) fixpoint of a functor equation and provides a suitable universe for reasoning about behavioural issues. In this context, final coalgebras are called *coinductive* or *left datatypes* in [Hag87b] or [CS92], *codatatypes* in [Kie98b, Kie98a], *final systems* in [Rut00] or *object types* in [Jac96].

Morphisms and bissimulation. Let (S, α_S) and (T, α_T) be two F-*coalgebras*, where F is an arbitrary functor. A function $f : S \rightarrow T$ is a *homomorphism* of F-*coalgebras*, or F-*homomorphism*, if $F(f) \cdot \alpha_s = \alpha_T \cdot f$. Pictorially,

$$\begin{tikzcd}
S \arrow[r, "f"] \arrow[d, "\alpha_S"'] & T \arrow[d, "\alpha_T"] \\
F(S) \arrow[r, "F(f)"'] & F(T)
\end{tikzcd}$$

Intuitively, homomorphisms are functions that preserve and reflect F-transition structures. The identity function on an F-coalgebra (S, α_s) is always a homomorphism, and the composition of two homomorphisms is again a homomorphism. Thus the collection of all F-coalgebras and corresponding homomorphisms forms a category, which is denoted by Set_F.

An F-homomorphism $f : S \rightarrow T$ with an inverse $f^{-1} : T \rightarrow S$ which is also a homomorphism is called an *isomorphism* between S and T. As usual, $S \cong T$ means that there exists an isomorphism between S and T. An injective homomorphism is called *monomorphism.* Dually, a surjective homomorphism is called *epimorphism.* Given systems S and T, we say that S can be *embedded* into T if there is a monomorphism from S to T. If there exists an epimorphism form S to T, T is called a *homomorphic image* of S. In that case, T is also called a *quotient* of S.

A *bisimulation* between two systems is intuitively, a transition structure respecting relation between sets os states. Formally, it is defined, for an arbitrary Set functor F as follows:

Let (S, α_S) and (T, α_T) be F-coalgebras. A subset $R \subseteq S \times T$ of the Cartesian product of S and T is called an F-*bisimulation* between S and T if there exists an F-transition structure $\alpha_R : R \rightarrow F(R)$ such that the projections form R to S and T are F-homomorphisms.

Pictorially,

$$\begin{tikzcd}
S \arrow[d, "\alpha_S"] & R \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[d, "\alpha_R"] & T \arrow[d, "\alpha_T"] \\
F(S) & F(R) \arrow[l, "F(\pi_1)"] \arrow[r, "F(\pi_2)"'] & F(T)
\end{tikzcd}$$

We also say, making explicit reference to the transition structures, that (R, α_R) is a bisimulation between (S, α_S) and (T, α_T). If $(T, \alpha_T) = (S, \alpha_S)$ then (R, α_R) is called a bisimulation on (S, α_S). A *bisimulation equivalence* is a bisimulation which is also an equivalence relation. Two states s and t are called *bisimilar* if there exists a bisimulation R with $\langle s, t\rangle \in R$.

Coinduction. Structures which are generated by a collection of constructors, like natural numbers (generated by zero and the successor function) or finite lists or trees, are classified as algebraic. Formally they are initial algebras. Induction is used both as a definition principle, and as a proof principle for such structures. Their duals are the coalgebraic structures, which do not come equipped with constructor operations but with what are sometimes called "destructor" operations (also called observers, accessors, transition maps, or mutators). [JR97]

Spaces of infinite data (including, for example, infinite lists, and non-well-founded sets) are generally of this kind. In general, dynamical systems with hidden, black-box state space, to which a user only has limited access via specified (observer or mutator) operations, are coalgebras of some kind. In the coalgebraic context *coinduction* is the appropriate technique both as a definition principle and as a proof principle. Proofs by coinduction equivales to finding an appropriate bisimulation relating the terms under consideration.

Consider, for example, functor $T\,X = A \times X$, where A is a fixed set. A coalgebra $U \longrightarrow T\,U$ consists of two functions $U \longrightarrow A$ and $U \longrightarrow U$, called $\mathtt{value} : U \longrightarrow A$ and $\mathtt{next} : U \longrightarrow U$. Given an element $u \in U$, we can either

1. produce an element in A, namely $\mathtt{value}(u)$;
2. produce a next element in U, namely $\mathtt{next}(u)$.

Steps 1. and 2. can be repeated, starting from any other element in U, namely $\mathtt{next}(u)$. By proceeding in this way we can get for each element $u \in U$ an infinite sequence $(\alpha_1, \alpha_2, ...) \in A^{\mathbb{N}}$ of elements of $\alpha_i = \mathtt{value}(\mathtt{next}^{(n)}(u)) \in A$. This sequence of elements that u gives rise to is what we can *observe* about u. Two elements u_1 and $u_2 \in U$ may well give rise to the same sequence of elements of A, without actually being equal as elements of U. In such a case one calls u_1 and u_2 observationally indistinguishable, or bisimilar.

The study of a broad notion of abstract data type, in the *initial algebraic* and *final coalgebraic* trends, dates back to the Adj group in the 1970's and later to T. Hagino landmark thesis [Hag87a]. As a research area it is now usually referred to as the theory of *categorical data types*. Generic combinators, parametric on the functor encoding the type signature, arise as universal arrows. They encode several recursion patterns which depend on the *shape* of the structure which the algorithm consumes, generates or, simply, 'rests on' (as a virtual data structure). In particular, *iteration* (respectively, *coiteration*) is modelled by the inductive (coinductive) extension of an algebra (coalgebra), *i.e.*, the unique arrow from (to) the initial (final) algebra (coalgebra). Such constructions become known in the area of *constructive algoritmics* [Mal90, Fok92b, BM97], as, respectively, *catamorphisms* and *anamorphisms*. A number of specialisations of such basic schemata have been proposed. L. Meertens introduced *paramorphisms* in [Mee92] which correspond to *primitive recursion*. The dual notion of *apomorphism*, that will be used in chapter 4, is due to T. Uustalu and V. Vene [VU97].

Such a research area, devoted to the development of program calculi directly based on, *i.e.*, actually driven by, type specifications, builds upon both the *genericity* and the *calculational style* entailed by category theory, the latter arising from the fact that most categorical properties can be formulated as (usually equational) laws. This has had a fundamental impact on algorithm derivation and transformation, mainly on the framework of *functional programming*. Around the end of the eighties, the so-called *Bird-Meertens formalism* [Bir87, BM87], originally an equational theory of sequences which formed the basis of a calculus for transforming list based programs, had emerged. R. Backhouse [Bac88] published on the basic role of category-theoretic universals in programming and G. Malcolm [Mal90] made the community aware of the foundational work of Hagino. Since then the area has known a remarkable progress, as witnessed by the vast bibliography published on both the theory and applications (see [BM97] as a textbook and [BJJM98] for a tutorial introduction).

Chapter 2

Stateless Software Connectors

Summary

This chapter introduces a purely relational model for software connectors which do not exhibit any form of internal memory (or local state). Therefore, a connector is modeled as a relation between values present at its input and output ports. Of course, only simultaneous observation of ports is allowed. Despite this restriction in expressiveness such connectors are still useful in a number of relevant applications. Moreover the model is both simple and intuitive. In particular, connector equivalence and refinement correspond to relational equality and inclusion, respectively. The chapter is based on publication [BB04a].

2.1 Synchronous stateless connectors

Connectors have *interface points*, or *ports*, through which messages flow. Each port has an *interaction polarity* (either *input* or *output*), but, in general, connectors are blind with respect to the data values flowing through them. Consequently, let us assume $\mathbb{D}$ as the generic type of such values.

Definition 2.1 (Connector) *Let C be a connector with m input and n output ports. Its semantics is given by a relation*

$$[\![C]\!] : \mathbb{D}^n \longleftarrow \mathbb{D}^m \tag{2.1}$$

We start by defining a number a number of basic connectors.

Synchronous channel

The most elementary connector is the *synchronous channel* with two ports of opposite polarity. Its semantics is simply the identity relation on the data domain $\mathbb{D}$:

$$[\![\bullet \longmapsto \bullet]\!] \;=\; \mathsf{Id}_{\mathbb{D}} \tag{2.2}$$

which forces input and output to become mutually blocking, in the sense that any of them must wait for the other to be completed. The synchronous channel, however, is just a special case, for $f = \mathsf{id}$, of a more generic connector with the ability to perform, in a systematic way, any kind of data conversion on the flow of messages. For any function $f : \mathbb{D} \longleftarrow \mathbb{D}$, the corresponding *transformer* is defined by

$$[\![\bullet \xmapsto{\ulcorner f \urcorner} \bullet]\!] \;=\; f \tag{2.3}$$

where f in the right hand side is the *relation* denoting the graph of *function* f.

Unreliable Channel

Any coreflexive relation over $\mathbb{D}$ provides a model for *unreliable channels*, *i.e.*, synchronous channels which may non-deterministically loose data.

$$[\![\bullet \xmapsto{\cdots} \bullet]\!] \;\subseteq\; \mathsf{Id}_{\mathbb{D}} \tag{2.4}$$

Filter

A *filter* is a channel in which some messages are discarded in a controlled way, according to a given predicate $\phi : \mathbf{2} \longleftarrow \mathbb{D}$. Noting that any predicate ϕ can be seen as a coreflexive $\Phi : \mathbb{D} \longleftarrow \mathbb{D}$ such that

$$d\Phi d' \;\text{ iff }\; d = d' \,\wedge\, (\phi\, d)$$

define

$$[\![\bullet \xmapsto{\ulcorner \phi \urcorner} \bullet]\!] \;=\; \Phi \tag{2.5}$$

Sources and Sinks

For each value $d \in \mathbb{D}$, a *source* $\Diamond_d$ is a device which permanently outputs d. It has only one output port, therefore, $[\![\Diamond_d]\!] : \mathbb{D} \longleftarrow \mathbf{1}$. Formally,

$$[\![\Diamond_d]\!] \quad = \quad \underline{d} \tag{2.6}$$

which defines the semantics of a *source* as a point in data domain $\mathbb{D}$. In pointwise notation,

$$[\![\Diamond_d]\!] \quad = \quad \{(d, *)\} \tag{2.7}$$

Alternatively, source can be represented graphically by

$$\bullet \xleftarrow{d} \bullet$$

Dually, a *sink* has only an input port which accepts, and discards, any possible message. Therefore, $[\![\blacklozenge]\!] : \mathbf{1} \longleftarrow \mathbb{D}$ is given by

$$[\![\blacklozenge]\!] \ = \ !_{\mathbb{D}} \tag{2.8}$$

i.e., by relation $\{(*, x) |\ x \in \mathbb{D}\}$, corresponding to the embedding of the universal final arrow in the category Set os sets and functions into the category Rel of sets and relations taken in this chapter as our universe of discourse.

An alternative graphic representation for sinks is

$$\bullet \relbar\joinrel\relbar\joinrel\dashv \bullet$$

Drain

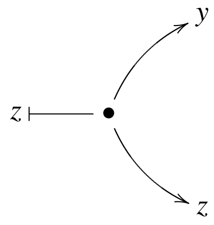
A *drain* $[\![\ \bullet \vdash\joinrel\relbar\joinrel\relbar\joinrel\dashv \bullet\ ]\!] \ : \ \mathbf{1} \longleftarrow (\mathbb{D} \times \mathbb{D})$ has two input, but no output, ports. This means that every message dropped at one of its ports is simply lost. A drain is *synchronous* if both write operations are requested to succeed at the same time (which implies that each write attempt remains pending until another write occurs in the other end-point). Formally, its definition is similar to that of the *sink* connector given above

$$[\![\ \bullet \vdash\joinrel\relbar\joinrel\relbar\joinrel\dashv \bullet\ ]\!] \ = \ !_{\mathbb{D} \times \mathbb{D}} \tag{2.9}$$

i.e., going pointwise, to relation $\{(*, (x, y)) |\ x, y \in \mathbb{D}\}$.

Broadcaster

The *broadcaster* connector replicates in each of its two output ports, any input received in its (unique) entry. The broadcaster $\lhd$ is depicted as follows:

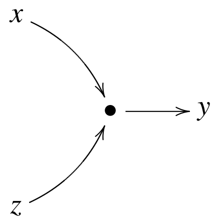


its semantics is, therefore, given by the diagonal relation $\triangle_{\mathbb{D}}$: $(\mathbb{D} \times \mathbb{D}) \longleftarrow \mathbb{D}$ on $\mathbb{D}$, defined by $\langle y, z \rangle \triangle_{\mathbb{D}} x \quad \equiv \quad x = y = z$:

$$[\![\lhd]\!] \;=\; \triangle_{\mathbb{D}} \tag{2.10}$$

Concentrator

The dual of the *broadcaster* is the *concentrator* which accepts identical messages in both of its input ports to be delivered on its unique output.



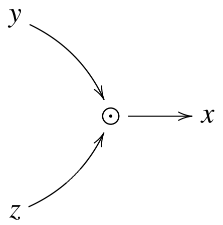
Formally,

$$[\![\rhd]\!] \;=\; [\![\lhd]\!]^{\circ} \tag{2.11}$$

Selective gateway

The *selective gateway* may be considered a variation of a *concentrator* connector: it does not impose uniformity on input, but on receiving two different values in its two different ports, it chooses one of them to be passed away through the output port. Such choice is non deterministic.

The connector is depicted as a *concentrator*



but with a totally different semantics:

$$[\![\blacktriangleright]\!] \;=\; \pi_1 \cup \pi_2 \tag{2.12}$$

It is instructive to compute the corresponding pointwise expression:

$$\begin{aligned} x\,(\pi_1 \cup \pi_2)\,\langle y, z\rangle \;&\equiv\; x(\mathsf{id}^\circ \cdot \pi_1)(y,z) \vee x(\mathsf{id}^\circ \cdot \pi_2)(y,z) \\ &\equiv\; x = \pi_1(y,z) \vee x = \pi_2(y,z) \\ &\equiv\; x = y \vee x = z \end{aligned}$$

Selective broadcaster

The converse of a *selective gateway* is the *selective broadcaster* which selects non deterministically one of its output ports. Formally,

$$[\![\blacktriangleleft]\!] \;=\; [\![\blacktriangleright]\!]^\circ \tag{2.13}$$

Note that this connector in not admissible in Reo [Arb03, Arb04] whose semantics takes as a basic assumption that all data is dispatched in all available output ports (*cf.*, the *pumping station* metaphor).

2.2 New connectors from old

Aggregation

The typical aggregation scheme for connectors is *parallel* composition — putting connectors side by side without any interaction between them. The formal semantics is given by relational *product*, in the general case, or relational *split*, when both connectors have identical input signatures,

$$[\![C_1 \boxtimes C_2]\!] \;=\; [\![C_1]\!] \times [\![C_2]\!] \tag{2.14}$$

$$[\![\langle C_1, C_2\rangle]\!] \;=\; \langle [\![C_1]\!], [\![C_2]\!]\rangle \tag{2.15}$$

where $\times$ and $\langle\ ,\ \rangle$ are both *relators* in Rel. Recall that a relator is an endofunctor in Rel which, additionally, preserves inclusion $\subseteq$ and commutes with converse. Relational product is defined as $R \times S \;=\; \langle R \cdot \pi_1, S \cdot \pi_2\rangle$, where the *split* combinator is given by $\langle R, S\rangle = \pi_1^\circ \cdot R \cap \pi_2^\circ \cdot S$.

Computing their pointwise definition may help to build up one's intuition. For example,

$$\begin{aligned}
\langle y,z\rangle\langle [\![C_1]\!], [\![C_2]\!]\rangle x &= \langle y,z\rangle(\pi_1^\circ \cdot [\![C_1]\!] \cap \pi_2^\circ \cdot [\![C_2]\!])x \\
&= \langle y,z\rangle(\pi_1^\circ \cdot [\![C_1]\!])x \;\wedge\; \langle y,z\rangle(\pi_2^\circ \cdot [\![C_2]\!])x \\
&= \pi_1\langle y,z\rangle\, [\![C_1]\!]\, x \;\wedge\; \pi_2\langle y,z\rangle\, [\![C_2]\!] x \\
&= y[\![C_1]\!]x \;\wedge\; z[\![C_2]\!]x
\end{aligned}$$

Using *split* one may, for example, built a *broadcaster* out of two *synchronous channels*:

$$\begin{aligned}
& \langle [\![\ \bullet \longmapsto \bullet\ ]\!], [\![\ \bullet \longmapsto \bullet\ ]\!]\rangle \\
= & \qquad \{\ \text{semantics of synchronous channel}\} \\
& \langle \mathsf{id}_{\mathbb{D}}, \mathsf{id}_{\mathbb{D}}\rangle \\
= & \qquad \{\ \triangle\ \text{definition}\ \} \\
& \triangle_{\mathbb{D}} \\
= & \qquad \{\ \text{semantics of broadcaster}\} \\
& [\![\lhd]\!]
\end{aligned}$$

As expected, the $\boxtimes$ operator inherits from Rel the properties of relational product. In particular, it is *associative* and *commutative*.

Hook

The basic way of combining two connectors is by plugging the output ports of one of them to the inputs of the other. Semantically, this amounts to relational composition:

$$[\![C_1\,;C_2]\!] \;=\; [\![C_2]\!] \cdot [\![C_1]\!] \tag{2.16}$$

for C_1, C_2 with matching signatures.

In the general case, however, this form of composition has to be made partial, *i.e.*, connecting only a (specified) subset of ports. This is achieved by the *hook* combinator which encodes a *feedback* mechanism, drawing a direct connection between an output and an input port. Formally, $[\![\mathbb{C} \Lsh^j_i]\!]$ is obtained from $[\![\mathbb{C}]\!]$, by deleting references to ports i and j. Formally,

$$q'([\![\mathbb{C} \Lsh^j_i]\!])q \text{ iff } \langle \exists\, t, t' \;::\; t'[\![\mathbb{C}]\!]t \wedge q' = t'|_i \wedge q = t|_j \wedge t'_{\#i} = t_{\#j} \rangle \tag{2.17}$$

where $t|_i$ and $t_{\#i}$ represents, respectively, a tuple of data values t from which the data corresponding to port i has been deleted, and the tuple component corresponding to such data. Clearly, $R \Lsh^j_i \colon \mathbb{D}^{m-1} \longleftarrow \mathbb{D}^{n-1}$ if $R : \mathbb{D}^m \longleftarrow \mathbb{D}^n$.

Sequential composition in the sense of definition (2.16) is retrieved as a special case. Actually, suppose $\mathbb{C}_1$ has a unique output port labelled o whereas connector $\mathbb{C}_2$ has a unique port input port i. Then,

$$\begin{aligned}
& q'([\![(\mathbb{C}_1 \boxtimes \mathbb{C}_2) \Lsh^o_i]\!])q \\
\equiv \quad & \quad \{ \text{ \textit{hook} definition} \} \\
& \langle \exists\, t, t' \;::\; t'[\![(\mathbb{C}_1 \boxtimes \mathbb{C}_2)]\!]t \wedge q' = t'|_i \wedge q = t|_o \wedge t'_{\#i} = t_{\#o} \rangle \\
\equiv \quad & \quad \{ \text{ semantics of aggregation} \} \\
& \langle \exists\, t, t' \;::\; t'([\![\mathbb{C}_1]\!] \times [\![\mathbb{C}_2]\!])t \wedge q' = t'|_i \wedge q = t|_o \wedge t'_{\#i} = t_{\#o} \rangle \\
\equiv \quad & \quad \{ \text{ signatures of } [\![\mathbb{C}_1]\!] \text{ and } [\![\mathbb{C}_2]\!] \} \\
& \langle \exists\, c \;::\; (c, q')([\![\mathbb{C}_1]\!] \times [\![\mathbb{C}_2]\!])(q, c) \rangle \\
\equiv \quad & \quad \{ \text{ relational product } \} \\
& \langle \exists\, c \;::\; c[\![\mathbb{C}_1]\!]q' \;\wedge\; q[\![\mathbb{C}_2]\!]c \rangle \\
\equiv \quad & \quad \{ \text{ relational composition } \} \\
& q'([\![\mathbb{C}_1]\!] \cdot [\![\mathbb{C}_2]\!])q \\
\equiv \quad & \quad \{ \text{ definition (2.16) } \} \\
& q'[\![C_1 \,;\, C_2]\!]q
\end{aligned}$$

Join.

The effect of this combinator is to plug ports with identical polarity. The aggregation of input ports is done by a *right join* ($\mathbb{C}\ {}^{i}_{j} > z$), where $\mathbb{C}$ is a connector, and i and j are ports and z is a fresh name used to identify the new port. Port z receives asynchronously messages sent by either i or j. When messages are sent at same time the combinator chooses one of them in a nondeterministic way. On the other hand, aggregation of *output ports* resorts to a *left join* ($z <^{i}_{j} \mathbb{C}$). This behaves like a *broadcaster* sending synchronously messages from z to both i and j.

Formally, given a connector $[\![\mathbb{C}]\!] : \mathbb{D}^n \longleftarrow \mathbb{D}^m$, a right join over $\mathbb{C}$ is specified by

$$q'([\![\mathbb{C}\,{}^{i}_{j} > z]\!])q \text{ iff } \langle \exists\, t \;::\; t[\![\mathbb{C}]\!]q \wedge q'|_z = t|_{i,j} \wedge (q'_{\#z} = t_{\#i} \vee q'_{\#z} = t_{\#j})\rangle \tag{2.18}$$

Clearly, $[\![\mathbb{C}\,{}^{i}_{j} > z]\!] : \mathbb{D}^{n-1} \longleftarrow \mathbb{D}^m$.

The definition of a left join is similarly made in terms of a relation $[\![z <^{i}_{j} \mathbb{C}]\!] : \mathbb{D}^n \longleftarrow \mathbb{D}^{m-1}$ given by

$$q'([\![z <^{i}_{j} \mathbb{C}]\!])q \text{ iff } \langle \exists\, t \;::\; q'[\![\mathbb{C}]\!]t \wedge q|_z = t|_{i,j} \wedge q_{\#z} = t_{\#i} = t_{\#j}\rangle \tag{2.19}$$

Lemma 2.1 *The order in which ports are plugged is irrelevant. In particular*

$$\mathbb{C}\,{}^{i}_{j} > z \;=\; \mathbb{C}\,{}^{j}_{i} > z \tag{2.20}$$

$$(\mathbb{C}\,{}^{i}_{j} > z)\,{}^{z}_{k} > v \;=\; (\mathbb{C}\,{}^{i}_{k} > z)\,{}^{z}_{j} > v \;=\; (\mathbb{C}\,{}^{k}_{j} > z)\,{}^{z}_{i} > v \tag{2.21}$$

$$z <^{i}_{j} \mathbb{C} \;=\; z <^{j}_{i} \mathbb{C} \tag{2.22}$$

$$v <^{z}_{k} (z <^{i}_{j} \mathbb{C}) \;=\; v <^{z}_{j} (z <^{i}_{k} \mathbb{C}) \;=\; v <^{z}_{i} (z <^{k}_{j} \mathbb{C}) \tag{2.23}$$

Proof. The proofs are obtained by unfolding the definitions. For (2.20) we reason

$$\begin{array}{ll}
& \mathbb{C}\,{}^{i}_{j} > z \\
\equiv & \quad \{ \text{ signature of } \mathbb{C} \} \\
& q'([\![\mathbb{C}\,{}^{i}_{j} > z]\!])q \\
\equiv & \quad \{ \text{ \textit{right join} definition} \} \\
& \langle \exists\, t \;::\; t[\![\mathbb{C}]\!]q \wedge q'|_z = t|_{i,j} \wedge (q'_{\#z} = t_{\#i} \vee q'_{\#z} = t_{\#j})\rangle \\
\equiv & \quad \{ \; t|i, j = t|j, i \text{ and } \vee \text{ commutative } \} \\
& \langle \exists\, t \;::\; t[\![\mathbb{C}]\!]q \wedge q'|_z = t|_{j,i} \wedge (q'_{\#z} = t_{\#j} \vee q'_{\#z} = t_{\#i})\rangle \\
\equiv & \quad \{ \text{ definition (2.18)} \} \\
& q'([\![(\mathbb{C}\,{}^{j}_{i} > z)]\!])q \\
\equiv & \quad \{ \text{ \textit{right join} definition } \} \\
& \mathbb{C}\,{}^{j}_{i} > z
\end{array}$$

Similarly, for (2.22)

$$
\begin{array}{ll}
 & z <^{i}_{j} \mathbb{C} \\
\equiv & \qquad \{ \text{ signature of } \mathbb{C} \} \\
 & q'(\![z <^{i}_{j} \mathbb{C}]\!)q \\
\equiv & \qquad \{ \text{ \textit{left join} definition } \} \\
 & \langle \exists\, t \ :: \ q' [\![\mathbb{C}]\!] t \wedge q|_z = t|_{i,j} \wedge q_{\#z} = t_{\#i} = t_{\#j} \rangle \\
\equiv & \qquad \{ \ t|i, j = t|j, i \text{ and } \wedge \text{ commutative } \} \\
 & \langle \exists\, t \ :: \ q' [\![\mathbb{C}]\!] t \wedge q|_z = t|_{j,i} \wedge q_{\#z} = t_{\#j} = t_{\#i} \rangle \\
\equiv & \qquad \{ \text{ definition (2.19)} \} \\
 & q'(\![z <^{j}_{i} \mathbb{C}]\!)q \\
\equiv & \qquad \{ \text{ \textit{left join} definition} \} \\
 & z <^{j}_{i} \mathbb{C}
\end{array}
$$

□

2.3 An example

This section illustrates through an example the use of stateless connectors to build useful coordination patterns. They are, of course, a subset of what can be expressed in Reo, in the sense that no *local state* nor *context* information is considered in this model. Such issues will be revisited, however, in chapters 3 and 5, respectively.

The pattern depicted in Fig. 2.3 assures that the flow of messages in a synchronous channel σ is externally controlled. This means that a control signal, produced by an external source, is required for a received message to be delivered at the output end-point. Within the model introduced above this is achieved by directing messages to the input of a *broadcaster* whose output ports are connected to *synchronous channel* and to a *synchronous drain* which, on its turn, accepts the control signals. Formally, the pattern is given by the following expression in the connector algebra:

$$(\lhd \ \boxtimes \bullet\!\longmapsto\!\bullet \ \boxtimes \ \bullet\!\vdash\!\!\!\relbar\!\!\!\relbar\!\!\!\dashv\!\bullet\,) \Lsh^{m,n}_{i,f} \tag{2.24}$$

The intuition on the correctness of this scheme is that, because, both the outputs of the *broadcaster* and the two end-points of the *drain* are synchronized, the read

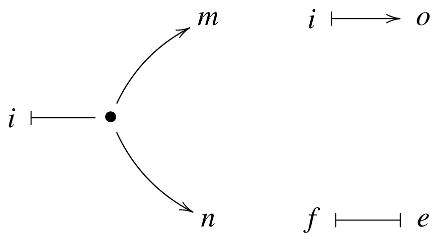


Figure 2.1: The *External control flow* pattern.

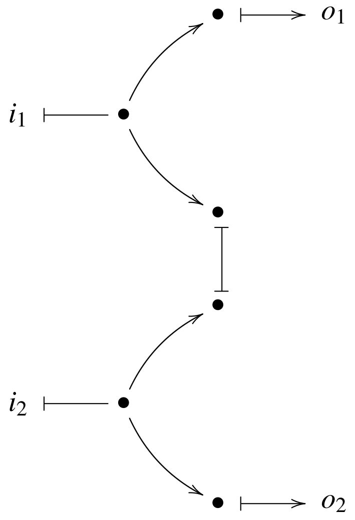


Figure 2.2: A synchronization barrier

operation on channel σ is completed simultaneously with the writing of the control signal on the free end-point of the *drain*. The reason for choosing a *drain* is simply that the actual contents of control messages is irrelevant in this context.

A generalization of this pattern is shown in Fig. 2.3. This pattern, quite popular in the the coordination literature, is known as a *synchronization barrier*, aiming at enforcing of mutual synchronization between two channels. Formally it is described by the following expression as the reader may confirm,

$$(\lhd \boxtimes \lhd) \; ; \; ((\bullet \longmapsto \bullet) \boxtimes (\bullet \vdash\!\!\relbar\!\!\dashv \bullet) \boxtimes (\bullet \longmapsto \bullet)) \tag{2.25}$$

2.4 Towards a connector calculus

As glimpsed above, an algebra of connectors begins to emerge in which a variety of coordination patterns can be expressed. Moreover connectors in this model enjoy a number of properties generically applicable to reason and transform such patterns. Their validity is easily established by simple computations within the relational calculus.

2.4.1 The equational fragment

Relational equality provides the basic comparison tool for connectors, resulting in rich equational calculus. Let us look briefly into some of its laws.

- First notice that a *synchronous channel* acts as the identity for connector composition,

  $$(C \;;\; \bullet\!\longmapsto\!\bullet\,) \;=\; C \;=\; (\,\bullet\!\longmapsto\!\bullet \;;\, C) \tag{2.26}$$

  for C with a matching signature. As **;** inherits associativity from composition in Rel, the algebra has, at least, the structure of a category.

  Proof.

  $$\begin{array}{ll}
  & \llbracket C \;;\; \bullet\!\longmapsto\!\bullet \rrbracket \\
  = & \quad \{\ \text{relation composition (2.16)}\} \\
  & \llbracket \bullet\!\longmapsto\!\bullet \rrbracket \cdot \llbracket C \rrbracket \\
  = & \quad \{\ \text{definition (2.2)}\ \} \\
  & \mathsf{Id}_{\mathbb{D}} \cdot \llbracket C \rrbracket \\
  = & \quad \{\ \mathsf{Id}\ \text{is the left unit of composition}\ \} \\
  & \llbracket C \rrbracket \\
  = & \quad \{\ \mathsf{Id}\ \text{is the right unit of composition}\ \} \\
  & \llbracket C \rrbracket \cdot \mathsf{Id}_{\mathbb{D}} \\
  = & \quad \{\ \text{definition (2.2)}\ \} \\
  & \llbracket C \rrbracket \cdot \llbracket \bullet\!\longmapsto\!\bullet \rrbracket \\
  = & \quad \{\ \text{relation composition (2.16)}\ \} \\
  & \llbracket \bullet\!\longmapsto\!\bullet \;;\; C \rrbracket
  \end{array}$$

- Similarly, an *unreliable channel* acts as an absorbing element for sequential composition with any kind of synchronous channels σ (including *filters*):

$$(\bullet \xmapsto{\cdots} \bullet \; ; \; \bullet \xmapsto{\sigma} \bullet) \;=\; (\bullet \xmapsto{\sigma} \bullet \; ; \; \bullet \xmapsto{\cdots} \bullet) \;=\; \bullet \xmapsto{\cdots} \bullet \tag{2.27}$$

Proof. The proof is directly obtained from the law (2.26) above:

$$\begin{array}{ll}
 & \llbracket \bullet \xmapsto{\cdots} \bullet \; ; \; \bullet \xmapsto{\sigma} \bullet \rrbracket \\
= & \quad \{ \text{ definitions (2.26) and (2.16)} \} \\
 & \mathsf{Id}_{\mathbb{D}} \cdot \llbracket \bullet \xmapsto{\cdots} \bullet \rrbracket \\
= & \quad \{ \; \mathsf{Id}_{\mathbb{D}} \text{ is the identity for relational composition} \} \\
 & \llbracket \bullet \xmapsto{\cdots} \bullet \rrbracket \cdot \mathsf{Id}_{\mathbb{D}} \\
= & \quad \{ \text{ definition (2.26)} \} \\
 & \llbracket \bullet \xmapsto{\sigma} \bullet \; ; \; \bullet \xmapsto{\cdots} \bullet \rrbracket
\end{array}$$

- The expected behaviour of *transformers* is stated by the following law

$$(\bullet \xmapsto{\ulcorner f \urcorner} \bullet \; ; \; \bullet \xmapsto{\ulcorner g \urcorner} \bullet) \;=\; \bullet \xmapsto{\ulcorner g \cdot f \urcorner} \bullet \tag{2.28}$$

Proof.

$$\begin{array}{ll}
 & \llbracket \bullet \xmapsto{\ulcorner f \urcorner} \bullet \rrbracket \; ; \; \llbracket \bullet \xmapsto{\ulcorner g \urcorner} \bullet \rrbracket \\
= & \quad \{ \text{ relational composition (2.16)} \} \\
 & \llbracket \bullet \xmapsto{\ulcorner g \urcorner} \bullet \rrbracket \cdot \llbracket \bullet \xmapsto{\ulcorner f \urcorner} \bullet \rrbracket \\
= & \quad \{ \text{ definition (2.3)} \} \\
 & c \cdot f \\
= & \quad \{ \text{ (2.28)} \} \\
 & \llbracket \bullet \xmapsto{\ulcorner g \cdot f \urcorner} \bullet \rrbracket
\end{array}$$

- The behaviour of *filters* composition is given by the law

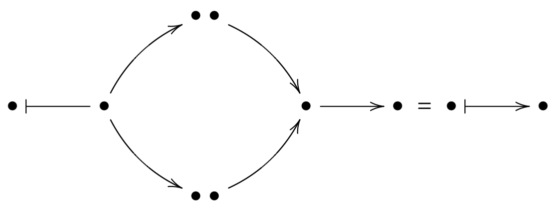
$$(\bullet \xmapsto{\ulcorner \phi \urcorner} \bullet \; ; \; \bullet \xmapsto{\ulcorner \psi \urcorner} \bullet) \;=\; \bullet \xmapsto{\ulcorner \phi \wedge \psi \urcorner} \bullet \tag{2.29}$$

Proof.

$$\begin{aligned}
& [\![\bullet \xmapsto{\ulcorner \phi \urcorner} \bullet \ ; \ \bullet \xmapsto{\ulcorner \psi \urcorner} \bullet]\!] \\
= \quad & \qquad \{ \text{ (2.16) and (2.5)} \} \\
& \Psi \cdot \Phi \\
= \quad & \qquad \{ \text{ union of coreflexives is intersection} \} \\
& \Psi \cap \Phi \\
= \quad & \qquad \{ \text{(2.5)} \} \\
& [\![\bullet \xmapsto{\ulcorner \phi \wedge \psi \urcorner} \bullet]\!]
\end{aligned}$$

- A *synchronous channel* can be implemented by the composition of a *broadcaster* and a *concentrator*:

$$(\lhd \ ; \ \rhd) \quad = \quad \bullet \longmapsto \bullet \tag{2.30}$$



Proof.

$$\begin{aligned}
& [\![\lhd \ ; \ \rhd]\!] \\
= \quad & \qquad \{ \text{ (2.16)} \} \\
& [\![\rhd]\!] \ \cdot \ [\![\lhd]\!] \\
= \quad & \qquad \{ \text{ definitions (2.11) and (2.10) } \} \\
& \triangle_{\mathbb{D}} \ \cdot (\triangle_{\mathbb{D}})^{\circ} \\
= \quad & \qquad \{ \text{ identity} \} \\
& \mathsf{Id}_{\mathbb{D}} \\
= \quad & \qquad \{ \text{ definition (2.2) } \} \\
& [\![\ \bullet \longmapsto \bullet \ ]\!]
\end{aligned}$$

- Curiously, replacing a *concentrator* by a *merger* also produces a synchronous channel.

$$(\lhd \ ; \ \blacktriangleright) \quad = \quad \bullet \longmapsto \bullet \tag{2.31}$$

Proof.

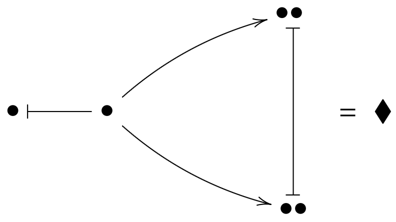
$$\begin{array}{cl}
 & [\![\lhd \ ; \ \blacktriangleright]\!] \\
= & \qquad \{\ (2.16)\} \\
 & [\![\blacktriangleright]\!] \ \cdot \ [\![\lhd]\!] \\
= & \qquad \{\ \text{definitions (2.12) and (2.10)}\ \} \\
 & (\pi_1 \cup \pi_2) \cdot \triangle_{\mathbb{D}} \\
= & \qquad \{\ \cdot R \text{ is lower adjoint in } \cdot R \dashv /R\text{, thus preserving colimits}\} \\
 & \pi_1 \cdot \triangle_{\mathbb{D}} \ \cup \ \pi_2 \cdot \triangle_{\mathbb{D}} \\
= & \qquad \{\ \text{definition of } \triangle_{\mathbb{D}} \text{ and product cancelation}\} \\
 & \mathsf{Id}_{\mathbb{D}} \cup \mathsf{Id}_{\mathbb{D}} \\
= & \qquad \{\ \text{identity}\} \\
 & \mathsf{Id}_{\mathbb{D}} \\
= & \qquad \{\ \text{definition} \quad (2.2)\ \} \\
 & [\![\ \bullet \longmapsto \bullet\ ]\!]
\end{array}$$

Similarly,

$$(\blacktriangleleft \ ; \ \blacktriangleright) \quad = \quad \bullet \longmapsto \bullet \tag{2.32}$$

- Sink can be realized by the composition of a broadcaster and a synchronous drain.

$$\lhd \ ; \bullet \vdash\!\!\!-\!\!\!-\!\!\!\dashv \bullet \quad = \quad \blacklozenge \tag{2.33}$$

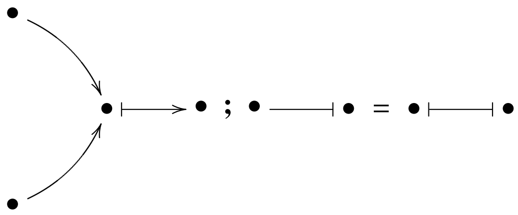


Proof.

$$
\begin{array}{cl}
 & [\![\lhd \;\; ; \;\; \bullet \vdash\!\!\relbar\!\!\relbar\!\!\dashv \bullet \,]\!] \\
= & \qquad \{ \text{ relational composition (2.16)} \} \\
 & [\![\, \bullet \overset{\blacktriangledown}{\vdash\!\!\relbar\!\!\relbar\!\!\dashv} \bullet \,]\!] \; \cdot \; [\![\lhd]\!] \\
= & \qquad \{ \text{ definitions (2.9) and (2.10) } \} \\
 & !_{\mathbb{D}\times\mathbb{D}} \cdot \, \triangle_{\mathbb{D}} \\
= & \qquad \{ \text{ ! universal: } ! \cdot f = ! \} \\
 & !_{\mathbb{D}} \\
= & \qquad \{ \text{ definition (2.8) } \} \\
 & [\![\blacklozenge]\!]
\end{array}
$$

- A symmetric law resorts to a concentrator and a sink to produce a drain, *i.e.*,

$$
\rhd \;\; ; \blacklozenge \quad = \quad \bullet \vdash\!\!\relbar\!\!\relbar\!\!\dashv \bullet \tag{2.34}
$$



Proof.

$$
\begin{array}{cl}
 & [\![\rhd \;\; ; \blacklozenge]\!] \\
= & \qquad \{ \text{ relational composition (2.16)} \} \\
 & [\![\blacklozenge]\!] \; \cdot \; [\![\rhd]\!] \\
= & \qquad \{ \text{ definitions (2.8) and (2.11) } \} \\
 & !_{\mathbb{D}} \cdot \, \triangle^{\circ}_{\mathbb{D}} \\
= & \qquad \{ \text{ ! universal: } ! \cdot f = ! \} \\
 & !_{\mathbb{D}\times\mathbb{D}} \\
= & \qquad \{ \text{ definition (2.9) } \} \\
 & [\![\, \bullet \vdash\!\!\relbar\!\!\relbar\!\!\dashv \bullet \,]\!]
\end{array}
$$

2.4.2 Connector refinement

In a model which specifies connectors as binary relations, refinement laws are stated in terms of relational inclusion. This means that for every input for which connector C is defined, all results delivered by C for this input are explicitly allowed by the refined connector R. On the other hand, if R is not defined for some input, then S also must not be defined for the same input. The situation is illustrated by the following two laws:

- A *lossy channel* is refined by a *synchronous channel*,

$$\bullet \overset{\cdots}{\longmapsto} \bullet \ \subseteq \ \bullet \longmapsto \bullet \tag{2.35}$$

  as any coreflexive is, by definition, a subset of the identity relation.

- The composition of a *concentrator* with *broadcaster*, which replicates on the output ports the pair of identical values received on input, is refined by a product of two *synchronous channels*,

$$(\rhd \ ; \ \lhd) \subseteq (\bullet \longmapsto \bullet \times \bullet \longmapsto \bullet) \tag{2.36}$$

Proof.

$$\begin{array}{cl}
 & [\![\rhd \ ; \ \lhd]\!] \\
= & \qquad \{\text{ relational composition (2.16)}\} \\
 & [\![\lhd]\!] \cdot [\![\rhd]\!] \\
= & \qquad \{\text{ definitions (2.11) and (2.10)}\} \\
 & \triangle_{\mathbb{D}}^{\circ} \cdot \triangle_{\mathbb{D}} \\
\subseteq & \qquad \{\text{definition of the diagonal relation}\} \\
 & \mathsf{Id}_{\mathbb{D}\times\mathbb{D}} \\
= & \qquad \{\text{functoriality}\} \\
 & \mathsf{Id}_{\mathbb{D}} \times \mathsf{Id}_{\mathbb{D}} \\
= & \qquad \{\text{definition (2.2) and product }\} \\
 & [\![(\bullet \longmapsto \bullet \times \bullet \longmapsto \bullet)]\!]
\end{array}$$

2.5 Extensions

2.5.1 Time-stamped connectors

In a number of common situations it becomes increasingly difficult to keep track of time constraints and a more precise setting is needed. In order to cope with situations like that the model is enriched with a (rather weak) notion of time. The model assumes that, on crossing the borders of a connector, every data value becomes labelled by a '*time stamp*' intended to express *order of occurrence*. As in [Arb03], temporal *simultaneity* is simply understood as *atomicity*, in the sense that two equally tagged input or output events are supposed to occur in an atomic way, that is, without being interleaved by other events.

Definition 1 *(Time-stamped connectors) Let C be a connector with m input and n output ports,* $\mathbb{D}$ *a generic type of values, and* $\mathbb{T}$ *a domain for time tags. Its semantics is given by a relation*

$$[\![C]\!] : (\mathbb{D} \times \mathbb{T})^n \longleftarrow (\mathbb{D} \times \mathbb{T})^m \tag{2.37}$$

where $\langle \mathbb{T}, \leq \rangle$ *is a total order acting as the domain of time tags.*

Note that the semantics of a connector

$$[\![C]\!] : (\mathbb{D} \times \mathbb{T}) \longleftarrow (\mathbb{D} \times \mathbb{T})$$

can be split into two relations: one, $\mathsf{data}.[\![C]\!] : \mathbb{D} \longleftarrow \mathbb{D}$, over the data values and another, $\mathsf{time}.[\![C]\!] : \mathbb{T} \longleftarrow \mathbb{T}$, over the time tags, as follows,

$$\begin{aligned} \mathsf{data}.[\![C]\!] &= \pi_1 \cdot [\![C]\!] \cdot \pi_1^\circ \\ \mathsf{time}.[\![C]\!] &= \pi_2 \cdot [\![C]\!] \cdot \pi_2^\circ \end{aligned}$$

This extends to the general case (2.37) in the obvious way

$$\mathsf{data}.[\![C]\!] = \Pi_{i=1..n}\pi_1 \cdot [\![C]\!] \cdot \Pi_{i=1..m}\pi_1^\circ \tag{2.38}$$

$$\mathsf{time}.[\![C]\!] = \Pi_{i=1..n}\pi_2 \cdot [\![C]\!] \cdot \Pi_{i=1..m}\pi_2^\circ \tag{2.39}$$

Within this extended model it is possible to specify some form of *asynchronous* connectors. This is illustrated in the sequel by the introduction of three such connectors: a *postponer*, which introduces delays, and asynchronous versions of a *drain* and a *broadcaster*.

Postponer

Temporal adjustments, through the introduction of delays, are often required in coordination situations. In this extended model, a *postponer* is specified as

$$\mathsf{data}.[\![\ \bullet \overset{\delta}{\longmapsto} \bullet\ ]\!] \;=\; \mathsf{Id}_{\mathbb{D}} \tag{2.40}$$

$$\mathsf{time}.[\![\ \bullet \overset{\delta}{\longmapsto} \bullet\ ]\!] \;=\; > \tag{2.41}$$

Asynchronous drain

In an *asynchronous drain* $\bullet \overset{\triangledown}{\longmapsto\!\dashv} \bullet\ : \mathbf{1} \longleftarrow (\mathbb{D} \times \mathbb{D})$ writing is non blocking. Just the opposite, the constraint is that the writing in both ports is not allowed to succeed at the same time. Formally,

$$\overline{\mathsf{time}.[\![\ \bullet \overset{\triangledown}{\longmapsto\!\dashv} \bullet\ ]\!]} \cap \mathsf{Id}_{\mathbb{T}} \;=\; \emptyset \tag{2.42}$$

where, in this context, given a relation $R : \mathbb{T} \times \mathbb{T} \longrightarrow \mathbf{1}$, relation $\overline{R} : \mathbb{T} \longrightarrow \mathbb{T}$ is given by

$$t'\overline{R}t \;\equiv\; *R\,(t', t)$$

Asynchronous broadcaster

Different from the synchronous version, the asynchronous broadcaster does not permit the simultaneous activation of its two output ports. Therefore, if at the data level both definitions coincide:

$$\mathsf{data}.[\![\lhd]\!] \;=\; [\![\lhd]\!] \;=\; \triangle_{\mathbb{D}} \tag{2.43}$$

the asynchronous semantics is imposed by the following temporal restriction:

$$=_{\mathbb{T}}\ \cdot \mathsf{time}.[\![\lhd]\!] \;=\; \mathsf{false} \tag{2.44}$$

where $=_{\mathbb{T}}$ is the equality predicate over $\mathbb{T}$.

2.5.2 Connectors with alternative ports

In the model discussed in the previous sections, neither the *concentrator* ($\rhd$) nor the *selective gateway* ($\blacktriangleright$) connector is able to capture the informal semantics typically associated to a *merger*. Actually, a merger is a connector which reads from either of its two input ports, with no specified order, and outputs the corresponding values. Clearly its domain is the disjoint sum $\mathbb{D} + \mathbb{D}$, but this contradicts the basic signature of a connector given in definition 2.1.

Let us relax that definition for a moment and admit coproducts in the signature of a connector. This enables the introduction of a new form of connector aggregation in alternative to parallel composition $\boxtimes$. Such an alternative form of aggregation resorts to relational *sum*, in the general case, or relational *either*, when both connectors have identical output signatures. I.e.,

$$[\![C_1 \boxplus C_2]\!] \quad = \quad [\![C_1]\!] + [\![C_2]\!] \tag{2.45}$$

$$[\![[C_1, C_2]]\!] \quad = \quad [[\![C_1]\!], [\![C_2]\!]] \tag{2.46}$$

where $+$ and $[\ ,\ ]$ are both *relators* in Rel, given by $R + S \ = \ [\iota_1 \cdot R, \iota_2 \cdot S]$ and $[R, S] = R \cdot \iota_1^\circ \cup S \cdot \iota_2^\circ$. Clearly, just as $\boxtimes$, $\boxplus$ inherits from Rel the properties of the underlying relational combinator. In particular, it is *associative* and *commutative*.

In this setting, the effect of merger can be easily obtained as the *either* of two *synchronous channels*. Formally,

$$\begin{array}{ll}
 & [[\![\bullet \longmapsto \bullet]\!], [\![\bullet \longmapsto \bullet]\!]] \\
\equiv & \qquad \{ \text{ semantics of synchronous channel} \} \\
 & [\mathsf{id}_{\mathbb{D}}, \mathsf{id}_{\mathbb{D}}] \\
\equiv & \qquad \{ \text{ [,] definition } \} \\
 & \mathsf{id}_{\mathbb{D}} \cdot \iota_1^\circ \cup \mathsf{id}_{\mathbb{D}} \cdot \iota_2^\circ \\
\equiv & \qquad \{ \text{ identity } \} \\
 & \iota_1^\circ \cup \iota_2^\circ
\end{array}$$

Notice that, by an argument similar to the one used in the proof of (2.34), another variant of an asynchronous drain, in which both simultaneous or non simultaneous input are allowed, can be obtained by the composition

$$[[\![\bullet \longmapsto \bullet]\!], [\![\bullet \longmapsto \bullet]\!]] \ ; \ \blacklozenge \tag{2.47}$$

We shall not, however, pursue this path, which would require further extensions to the definition of connector combinators. Instead, there are two equivalent decisions one may resort to achieve similar expressive power: either to define the type of a port as an eventually 0-ary sum of data values (in which case a merger would be just a synchronous channel with a single input port of type $\mathbb{D} + \mathbb{D}$), or to impose flat partial order structure to $\mathbb{D}$ so that the bottom value could stand for absence of information. In this last case the product of input ports will still be able to capture asynchronous arriving and dispatching of incoming data. A similar option, which explicitly takes into account *negative* information, will be considered in the model discussed in chapter 5.

Chapter 3

Stateful Software Connectors

Summary

This chapter introduces a new model for software connectors whose input-output behaviour is partially determined by a memory of past computations encoded as the connector's state space. Therefore each connector becomes a coalgebra for a functor capturing its signature of communication ports. Standard coalgebraic techniques apply; in particular connector equivalence boils down to bisimulation. In brief this models extends the one presented in chapter 2 in exactly the same sense that transition systems extend binary relations: coalgebras are just relations extended in time*. The chapter also discusses how some form of mobility can be expressed within the model, introducing a special connector, the* orchestrator*, whose role is to manage interconnection patterns that can change at run-time. The chapter is based on publications [BB04b] and [BB07].*

3.1 Introduction

A major limitation of the model introduced in the previous chapter is the impossibility of modelling connecters with buffering capacities. Those include, for example, a queue-based channel (known in Reo as a *fifo* channel) in which reading and writing are non mutually blocking operations.

A first attempt to accommodate in our repertoire connectors with some sort of asynchronous behaviour has already been suggested in the previous chapter: extending the elementary relational definition

$$[\![C]\!] : \mathbb{D}^n \longleftarrow \mathbb{D}^m$$

to

$$[\![C]\!] : (\mathbb{D} \times \mathbb{T})^n \longleftarrow (\mathbb{D} \times \mathbb{T})^m$$

under the assumption that, on crossing the borders of a connector, every data value becomes labelled by a *time stamp* of type $\mathbb{T}$. As then explained, $\mathbb{T}$ represents a (rather weak) notion of time intended to express *order of occurrence*.

Such an extension, however, is not enough to specify a channel obeying a strict *fifo* discipline. Actually, to express this kind of constraints in connector specification requires a fine-grain control over the flow of data; typical solutions resort to *infinite* data structures, for example *i.e.*, infinite sequences of messages and time stamps, as in [AR02, Arb03] or [BRS$^+$00].

The alternative model proposed in this chapter defines a connector not only in terms of input-output behaviour but also with respect to a memory of past computations encoded as an internal state space.

Let U be the type of such memory. A connector is then modelled as a relation involving not only the input and output domains, as before, but also U itself, that is

$$[\![C]\!] : \mathbb{D} \times U \longleftarrow U \times \mathbb{D} \tag{3.1}$$

which can also be represented, by transposition [OR04], by function

$$[\![C]\!] : \mathcal{P}(\mathbb{D} \times U) \longleftarrow U \times \mathbb{D} \tag{3.2}$$

or, in an equivalent way,

$$[\![C]\!] : \mathcal{P}(\mathbb{D} \times U)^{\mathbb{D}} \longleftarrow U \tag{3.3}$$

that is, in the form of a *coalgebra* ([Rut00, JR97]) for functor $\mathsf{F}X = \mathcal{P}(X \times \mathbb{D})^{\mathbb{D}}$.

The coalgebraic format is adequate to single out U as the connector internal state space, not externally available. For example, a channel with a single buffering capacity is modeled as a coalgebra over $U = \mathbb{D}$, whereas an unbounded buffer requires $U = \mathbb{D}^*$, where notation X^* stands, as usual, for finite sequences of X. In both cases a write operation updates the state variable and a read operation consumes it (or, respectively, its last element).

In this framework the queue-based channel mentioned above, to be referred in the sequel as the fifo_∞ connector, is given by the specification of two ports to which two operations over $\mathbb{D}^*$, corresponding to the reception and delivery of a $\mathbb{D}$ value, are associated. The rationale is that the operations are activated by the arrival of a data element (often referred to as a message) to the port. Formally,

$$\begin{aligned}
\mathsf{receive} &: \mathbb{D}^* \times \mathbb{D} \;\rightarrow\; \mathbb{D}^* \\
&= \mathsf{cons} \cdot \mathsf{swap} \\
\mathsf{deliver} &: \mathbb{D}^* \;\rightarrow\; \mathbb{D}^* \times (\mathbb{D} + \mathbf{1}) \\
&= \langle \mathsf{tl}, \mathsf{hd} \rangle
\end{aligned}$$

where swap is the product commutativity isomorphism. Grouping them together leads to a specification of the channel as an elementary transition structure over $\mathbb{D}^*$, *i.e.*, a pointed *coalgebra*

$$\langle [] \in \mathbb{D}^*, c : \mathbb{D}^* \longrightarrow (\mathbb{D}^* \times (\mathbb{D} + \mathbf{1}))^{(\mathbb{D}+\mathbf{1})} \rangle$$

where

$$\begin{aligned}
\overline{c} \;=\; \mathbb{D}^* \times (\mathbb{D} + 1) &\xrightarrow{\mathsf{dr}} \mathbb{D}^* \times \mathbb{D} + \mathbb{D}^* \xrightarrow{\mathsf{receive}+\mathsf{deliver}} \mathbb{D}^* + \mathbb{D}^* \times (\mathbb{D} + 1) \\
&\xrightarrow{\simeq} \mathbb{D}^* \times 1 + \mathbb{D}^* \times (\mathbb{D} + 1) \xrightarrow{[\mathsf{id} \times \iota_2, \mathsf{id}]} \mathbb{D}^* \times (\mathbb{D} + 1)
\end{aligned}$$

and the *seed* value $[] \in \mathbb{D}^*$ represents its initial state.

Note how this specification meets all the exogenous synchronization constraints, including the enforcing of a strict FIFO discipline. The temporal dimension, however, is no more explicit, but *built-in* in coalgebra dynamics.

The next four sections detail and exemplify this *connectors as coalgebras* model. Sections 3.6 introduces a special *orchestrator* connector to cope with some form of mobility and dynamic reconfiguration within the model.

3.2 Connectors as coalgebras

3.2.1 The general model

A software connector is specified by an interface which aggregates a number of *ports* represented by operations which regulate its behaviour. Each operation encodes the port reaction to a data item crossing the connector's boundary. Let U be the type of the connector's internal state space and $\mathbb{D}$ a generic data domain for messages, as before. In such a setting we single out three kinds of ports with the following signatures:

$$\mathsf{post} : U \longrightarrow U^{\mathbb{D}} \tag{3.4}$$

$$\mathsf{read} : U \longrightarrow (\mathbb{D} + 1) \tag{3.5}$$

$$\mathsf{get} : U \longrightarrow U \times (\mathbb{D} + 1) \tag{3.6}$$

where

- post is an input operation analogous to a write operation in conventional programming languages (see *e.g.*, [Arb02, PA98, Arb03]). Typically, a post port accepts data items and store them internally, in some form.
- read is a non-destructive output operation. This means that through a read port the environment might 'observe' a data item, but the connector's state space remains unchanged. Of course read is a partial operation, because there cannot be any guarantee that data is available for reading.
- get is a destructive variation of the read port. In this case the data item is not only made externally available, but also deleted from the connector's memory.

As mentioned above, connectors are formed by the aggregation of a number of post, read and get ports. According to their number and types one specific connector with well-defined behaviour may be defined. Let us consider some possibilities.

Sources and Sinks

The most elementary connectors are those with a unique port. According to its orientation they can be classified as

- Data *sources*, specified by a single read operation

  $$\Diamond_d \;=\; \langle d \in \mathbb{D}, \iota_1 : \mathbb{D} \longrightarrow \mathbb{D} + \mathbf{1} \rangle \tag{3.7}$$

  defined over state space $U = \mathbb{D}$ and initialized with value d.
- Data *sinks*, ie, connectors which are always willing to accept any data item, discarding it immediately. The state space of data sinks is irrelevant and, therefore, modeled by the singleton set $\mathbf{1} = \{*\}$. Formally,

  $$\blacklozenge \;=\; \langle * \in \mathbf{1}, ! : \mathbf{1} \longrightarrow \mathbf{1}^{\mathbb{D}} \rangle \tag{3.8}$$

  where ! is the (universal) map from any object to the (final) set $\mathbf{1}$.

Binary Connectors

Binary connectors are built by the aggregation of two ports, assuming the corresponding operations are defined over the same state space. This, in particular, enforces mutual execution of state updates.

- Consider, first, the aggregation of two read ports, denoted by read_1 and read_2, with possibly different specifications. Both of them are (non destructive) observers and, therefore, can be simultaneously offered to the environment. The result is a coalgebra simply formed by their *split*:

$$c \;=\; \langle u \in U, \langle \mathsf{read}_1, \mathsf{read}_2 \rangle : U \longrightarrow (\mathbb{D}+1) \times (\mathbb{D}+1) \rangle \tag{3.9}$$

- On the other hand, aggregating a post to a read port results in

$$c \;=\; \langle u \in U, \langle \mathsf{post}, \mathsf{read} \rangle : U \longrightarrow U^{\mathbb{D}} \times (\mathbb{D}+1) \rangle \tag{3.10}$$

- Replacing the read port above by a get one requires an additive aggregation to avoid the possibility of simultaneous updates leading to

$$c \;=\; \langle u \in U, \gamma_c : U \longrightarrow (U \times (\mathbb{D}+1))^{\mathbb{D}+\mathbf{1}} \rangle \tag{3.11}$$

  where[1]

$$\begin{aligned} \overline{\gamma_c} \;=\; U \times (\mathbb{D}+1) &\xrightarrow{\mathsf{dr}} U \times \mathbb{D} + U \xrightarrow{\overline{\mathsf{post}}+\mathsf{get}} U + U \times (\mathbb{D}+1) \\ &\xrightarrow{\simeq} U \times 1 + U \times (\mathbb{D}+1) \xrightarrow{[\mathsf{id}\times\iota_2,\mathsf{id}]} U \times (\mathbb{D}+1) \end{aligned}$$

  Channels of different kinds are connectors of this type. Recall the fifo_∞ example above: ports identified by $\mathsf{receive}$ and $\mathsf{deliver}$ have the same signature of a post and a get, respectively. An useful variant is the *filter* connector which discards some messages according to a given predicate $\phi : \mathbf{2} \longleftarrow \mathbb{D}$. The get port is given as before, *i.e.*, $\langle \mathsf{tl}, \mathsf{hd} \rangle$, but post becomes conditional on predicate ϕ, *i.e.*,

$$\mathsf{post} \;=\; \phi \longrightarrow \mathsf{cons} \cdot \mathsf{swap}, \pi_1$$

- A similar scheme is adopted for the combination of two post ports:

$$c \;=\; \langle u \in U, \gamma_c : U \longrightarrow U^{\mathbb{D}+\mathbb{D}} \rangle \tag{3.12}$$

  where

$$\begin{aligned} \overline{\gamma_c} = U \times (\mathbb{D}+\mathbb{D}) &\xrightarrow{\mathsf{dr}} U \times \mathbb{D} + U \times \mathbb{D} \\ &\xrightarrow{\overline{\mathsf{post}_1}+\overline{\mathsf{post}_2}} U + U \xrightarrow{\triangledown} U \end{aligned}$$

[1] In the sequel dr is the right distributivity isomorphism and $\triangledown$ the codiagonal function defined as the *either* of two identities, *i.e.*, $\triangledown = [\mathsf{id}, \mathsf{id}]$.

The General Case. Examples above lead to the specification of the following shape for a connector built by aggregation of P post, G get and R read ports:

$$c \;=\; \langle u \in U, \langle \gamma_c, \rho_c \rangle : U \longrightarrow (U \times (\mathbb{D}+1))^{P \times \mathbb{D} + G} \times (\mathbb{D}+1)^R \rangle \tag{3.13}$$

where ρ_c is the split R read ports, *i.e.*,

$$\rho_c : \; U \longrightarrow (\mathbb{D}+1) \times (\mathbb{D}+1) \times \ldots \times (\mathbb{D}+1) \tag{3.14}$$

and, γ_c collects ports of type post or get, which are characterized by the need to perform state updates, in the uniform scheme explained above for the binary case. Note that c is a coalgebra for functor

$$\mathsf{F}X \;=\; (X \times (\mathbb{D}+1))^{P \times \mathbb{D} + G} \times (\mathbb{D}+1)^R \tag{3.15}$$

which can be rewritten as

$$\mathsf{F}X \;=\; (\sum_{i \in P} X^{\mathbb{D}} + \sum_{j \in G} X \times (\mathbb{D}+1)) \times \prod_{k \in R} (\mathbb{D}+1) \tag{3.16}$$

The latter is, however, less amenable to symbolic manipulation in proofs.

3.3 Combinators

In the previous section, a general model of software connectors as pointed coalgebras was introduced and their construction by port aggregation discussed. To obtain descriptions of more complex interaction patterns, however, some forms of connector composition are needed. Such is the topic of the present section in which three combinators are defined: *concurrent composition*, *interleaving* and a generalisation of *pipelining* capturing arbitrary composition of post with either read or get ports.

3.3.1 Concurrent composition

Consider connectors c_1 and c_2 defined as

$$c_i = \langle u_i \in U_i, \langle \gamma_i, \rho_i \rangle : (U_i \times (\mathbb{D}+1))^{P_i \times \mathbb{D} + G_i} \times (\mathbb{D}+1)^{R_i} \rangle$$

with P_i ports of type post, R_i of type read and G_i of type get, for $i = 1, 2$. Their concurrent composition, denoted by $c_1 \boxdot c_2$ makes externally available all c_1 and c_2 *single* primitive ports, plus *composite* ports corresponding to the simultaneous activation of post (respectively, get) ports in the two operands. Therefore, $P' =$

$P_1 + P_2 + P_1 \times P_2$, $G' = G_1 + G_2 + G_1 \times G_2$ and $R' = R_1 + R_2$ become available in $c_1 \boxdot c_2$ as its interface sets. Formally, define

$$c_1 \boxdot c_2 : U' \longrightarrow (U' \times (\mathbb{D} + 1))^{P' \times \mathbb{D} + G'} \times (\mathbb{D} + 1)^{R'} \tag{3.17}$$

where

$$\begin{aligned}
&\overline{\gamma}_{c_1 \boxdot c_2} = \\
&U_1 \times U_2 \times (P_1 + P_2 + P_1 \times P_2) \times \mathbb{D} + (G_1 + G_2 + G_1 \times G_2) \xrightarrow{\simeq} \\
&(U_1 \times (P_1 \times \mathbb{D} + G_1) \times U_2 + U_1 \times U_2 \times (P_2 \times \mathbb{D} + G_2) + U_1 \times (P_1 \times \mathbb{D} + G_1) \times U_2 \times (P_2 \times \mathbb{D} + G_2) \\
&\xrightarrow{\gamma_1 \times \mathrm{id} + \mathrm{id} \times \gamma_2 + \gamma_1 \times \gamma_2} (U_1 \times (\mathbb{D} + 1)) \times U_2 + U_1 \times (U_2 \times (\mathbb{D} + 1)) + (U_1 \times (\mathbb{D} + 1)) \times (U_2 \times (\mathbb{D} + 1)) \\
&\xrightarrow{\simeq} U_1 \times U_2 \times (\mathbb{D} + 1) + U_1 \times U_2 \times (\mathbb{D} + 1) + U_1 \times U_2 \times (\mathbb{D} + 1)^2 \xrightarrow{\nabla + \mathrm{id}} \\
&U_1 \times U_2 \times (\mathbb{D} + 1) + U_1 \times U_2 \times (\mathbb{D} + 1)^2 \xrightarrow{\simeq} U_1 \times U_2 \times ((\mathbb{D} + 1) + (\mathbb{D} + 1))^2
\end{aligned}$$

and

$$\rho_{c_1 \boxdot c_2} = U_1 \times U_2 \xrightarrow{\rho_1 \times \rho_2} (\mathbb{D} + 1)^{R_1} \times (\mathbb{D} + 1)^{R_2} \xrightarrow{\simeq} (\mathbb{D} + 1)^{R_1 + R_2}$$

3.3.2 Interleaving

Interleaving is a restricted form of parallel composition which rules out the simultaneous execution of two ports, one in each of its operands. Formally the definition of $c_1 \boxplus c_2$ is obtained from that of $c_1 \boxdot c_2$ taking in (3.17) $P' = P_1 + P_2$ and $G' = G_1 + G_2$, and removing from the definition of $\overline{\gamma}_{c_1 \boxdot c_2}$ all references to composite terms, *i.e.*,

$$\begin{aligned}
\overline{\gamma}_{c_1 \boxplus c_2} &= U_1 \times U_2 \times (P_1 + P_2) \times \mathbb{D} + (G_1 + G_2) \xrightarrow{\simeq} \\
&\quad (U_1 \times (P_1 \times \mathbb{D} + G_1) \times U_2 + U_1 \times U_2 \times (P_2 \times \mathbb{D} + G_2) \\
&\quad \xrightarrow{\gamma_1 \times \mathrm{id} + \mathrm{id} \times \gamma_2} (U_1 \times (\mathbb{D} + 1)) \times U_2 + U_1 \times (U_2 \times (\mathbb{D} + 1)) \\
&\quad \xrightarrow{\simeq} (U_1 \times U_2 \times (\mathbb{D} + 1)) + (U_1 \times U_2 \times (\mathbb{D} + 1)) \\
&\quad \xrightarrow{\nabla} U_1 \times U_2 \times (\mathbb{D} + 1)
\end{aligned}$$

3.3.3 Hook

The *hook* combinator, $\Lsh^p_r$ plugs ports with opposite polarity within an arbitrary connector

$$c = \langle u \in U, \langle \gamma_c, \rho_c \rangle : U \longrightarrow (U \times (\mathbb{D} + 1))^{P \times \mathbb{D} + G} \times (\mathbb{D} + 1)^R \rangle$$

There are two possible plugging situations:

1. Plugging a post port p_i to a read r_j one, resulting in

$$\begin{aligned}
\rho_{c \Lsh^{p_i}_{r_j}} &= \langle r_1, \ldots, r_{j-1}, r_{j+1}, \ldots, r_R \rangle \\
\overline{\gamma}_{c \Lsh^{p_i}_{r_j}} &= U \times ((P-1) \times \mathbb{D} + G) \xrightarrow{\theta \times \mathsf{id}} U \times ((P-1) \times \mathbb{D} + G) \\
&\xrightarrow{\simeq} \textstyle\sum_{P-1} U \times \mathbb{D} + \sum_G U \xrightarrow{[p_1,\ldots,p_{i-1},p_{i+1},\ldots,p_p]+[g_1,\ldots,g_G]} \\
&U + U \times (\mathbb{D}+1) \xrightarrow{\simeq} U \times 1 + U \times (\mathbb{D}+1) \\
&\xrightarrow{[\mathsf{id} \times \iota_2, \mathsf{id}]} U \times (\mathbb{D}+1)
\end{aligned}$$

where $\theta : U \longrightarrow U$

$$\begin{aligned}
\theta &= U \xrightarrow{\triangle} U \times U \xrightarrow{\mathsf{id} \times r_j} U \times \mathbb{D} + 1 \\
&\xrightarrow{\simeq} U \times \mathbb{D} + U \xrightarrow{\overline{p_i} + \mathsf{id}} U + U \xrightarrow{\triangledown} U
\end{aligned}$$

2. Plugging a post port p_i to a get g_j one, resulting in

$$\begin{aligned}
\rho_{c \Lsh^{p_i}_{r_j}} &= \rho_c \\
\overline{\gamma}_{c \Lsh^{p_i}_{g_j}} &= U \times ((P-1) \times \mathbb{D} + (G-1)) \xrightarrow{\theta \times \mathsf{id}} \\
&U \times ((P-1) \times \mathbb{D} + (G-1)) \\
&\xrightarrow{\simeq} \textstyle\sum_{P-1} U \times \mathbb{D} + \sum_{G-1} U \\
&\xrightarrow{[p_1,\ldots,p_{i-1},p_{i+1},\ldots,p_p]+[g_1,\ldots,g_{j-1},g_{j+1},\ldots,g_G]} \\
&U + U \times (\mathbb{D}+1) \xrightarrow{\simeq} U \times 1 + U \times (\mathbb{D}+1) \\
&\xrightarrow{[\mathsf{id} \times \iota_2, \mathsf{id}]} U \times (\mathbb{D}+1)
\end{aligned}$$

where $\theta : U \longrightarrow U$

$$\begin{aligned}
\theta &= U \xrightarrow{g_j} U \times (\mathbb{D}+1) \xrightarrow{\simeq} U \times \mathbb{D} + U \\
&\xrightarrow{\overline{p_i} + \mathsf{id}} U + U \xrightarrow{\triangledown} U
\end{aligned}$$

Note that, according to the definition above, if the result of a reaction at a port of type read or get is of type **1**, which encodes the absence of any data item to be read, the associated post is not activated and, consequently, the interaction does not become effective.

Such unsuccessful read attempt can alternatively be understood as a *pending read request*. In this case the intended semantics for interaction with the associated post port is as follows: successive read attempts are performed until communication occurs. This version of *hook* is denoted by $\Rsh^p_r\ c$ and easily obtained by replacing, in the definition of θ above, step

$$U \times \mathbb{D} + U \xrightarrow{\overline{p_i}+\mathsf{id}} U + U$$

by

$$U \times \mathbb{D} + U \xrightarrow{\overline{p_i}+\theta} U + U$$

Both forms of the *hook* combinator can be applied to a whole sequence of pairs of opposite polarity ports, the definitions above extending as expected.

As done in the relational model discussed in the previous chapter, combinators *interleave* and *hook* can be put together to define a form of *sequential composition* in situations where all the post ports of the second operand (grouped in *in*) are connected to all the read and get ports of the first (grouped in *out*). It is assumed that hooks between two single ports extend smoothly to any product of ports (as arising from concurrent composition) in which they participate. Formally, we define by abbreviation

$$c_1 \mathbin{;} c_2 \overset{\text{abv}}{=} (c_1 \boxplus c_2) \Lsh^{in}_{out} \tag{3.18}$$

and

$$c_1 \bowtie c_2 \overset{\text{abv}}{=} \Rsh^{in}_{out} (c_1 \boxplus c_2) \tag{3.19}$$

3.4 Examples

3.4.1 Broadcasters and Mergers

Our first example is the *broadcaster*, a connector which replicates in each of its two (output) ports, any input received in its (unique) entry as depicted bellow. There are two variants of this connector denoted, respectively, by $\triangleleft$ and $\blacktriangleleft$. The first one corresponds to a *synchronous* broadcast, in the sense that the two get ports are activated simultaneously. The other one is *asynchronous*, in the sense that both of its get ports can be activated independently. Although we use the same symbol which, in the previous chapter, expresses the *selective broadcaster*, the connector is actually different, in the sense that in this *asynchronous* broadcaster all information arriving to the input port will eventually be present at both output ports. Also note this connector is also different from $\lhd$ defined by (2.43) and (2.44) which does not allow simultaneous output at both ports.

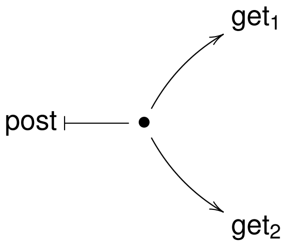


Figure 3.1: The *broadcaster* connector.

The definition of ◀ is rather straightforward as a coalgebra over $U = \mathbb{D} + \mathbf{1}$ and operations

$$\begin{aligned} \overline{\mathsf{post}} &: U \times \mathbb{D} \;\to\; U \;=\; \iota_1 \cdot \pi_2 \\ \mathsf{get}_1, \mathsf{get}_2 &: U \;\to\; U \times (\mathbb{D} + 1) \;=\; \triangle \end{aligned}$$

where $\triangle$ is the *diagonal* function, defined by $\triangle = \langle \mathsf{id}, \mathsf{id} \rangle$. The synchronous case, however, requires the introduction of two boolean flags initialized to $\langle \mathsf{false}, \mathsf{false} \rangle$ to witness the presence of get requests at both ports. The idea is that a value is made present at both the get ports if it has been previously received, as before, and there exists two reading requests pending. Formally, let $U = (\mathbb{D} + \mathbf{1}) \times (\mathcal{B} \times \mathcal{B})$ and define

$$\begin{aligned} \overline{\mathsf{post}} &: U \times \mathbb{D} \;\to\; U \;=\; \langle \iota_1 \cdot \pi_2, \pi_2 \cdot \pi_1 \rangle \\ \mathsf{get}_1 &: U \;\to\; U \times (\mathbb{D} + 1) \;=\; (=_* \cdot \pi_1 \;\to\; \langle \mathsf{id}, \pi_1 \rangle,\ \mathsf{getaux}_1) \end{aligned}$$

where

$$\mathsf{getaux}_1 \;=\; (\pi_2 \cdot \pi_2 \;\to\; \langle (\iota_2 \cdot \underline{*}) \times (\underline{\mathsf{false}} \times \underline{\mathsf{false}}), \pi_1 \rangle,\ \langle \mathsf{id} \times (\underline{\mathsf{true}} \times \mathsf{id}), \iota_2 \cdot \underline{*} \rangle)$$

I.e., if there is no information stored flag $*$ is returned and the state left unchanged. Otherwise, an output is performed but only if there is a previous request at the other port. If this is not the case the reading request is recorded at the connector's state. This definition precludes the possibility of a reading unless there are reading requests at both ports. The fact that both requests are served depends on their interaction with the associated post ports, *i.e.*, on the chosen hook discipline. The definition of get$_2$ is similar but for the boolean flags update:

$$\mathsf{getaux}_2 \;=\; (\pi_1 \cdot \pi_2 \;\to\; \langle (\iota_2 \cdot \underline{*}) \times (\underline{\mathsf{false}} \times \underline{\mathsf{false}}), \pi_1 \rangle,\ \langle \mathsf{id} \times (\mathsf{id} \times \underline{\mathsf{true}}), \iota_2 \cdot \underline{*} \rangle)$$

Dual to the *broadcaster* connector is the *merger* which concentrates messages arriving at any of its two post ports. The *merger*, denoted by $\rhd$, is similar to an asyn-

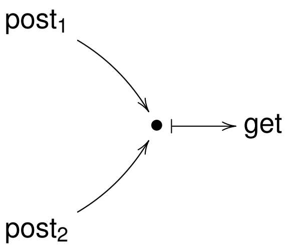


Figure 3.2: The *merger* connector.

chronous channel, as given in section 3.2, with two identical post ports. Another variant, denoted by ▶, accepts one data item a time, after which disables both post ports until get is activated. This connector is defined as a coalgebra over $U = \mathbb{D} + \mathbf{1}$ with

$$\begin{aligned}
\overline{\mathsf{post}_1} = \overline{\mathsf{post}_2} \;:&\; U \times \mathbb{D} \;\to\; U \\
=&\; (=_* \cdot\pi_1 \;\to\; \pi_1,\, \iota_1 \cdot \pi_2) \\
\mathsf{get} \;:&\; U \;\to\; U \times (\mathbb{D} + \mathbf{1}) \\
=&\; (=_* \to \langle \triangle, \mathsf{id}\rangle,\, \langle \iota_2 \cdot \underline{*}, \mathsf{id}\rangle)
\end{aligned}$$

3.4.2 Drains

A *drain* is a symmetric connector with two inputs, but no output, points. Operationally, every message arriving to an end–point of a drain is simply lost. A drain is *synchronous* when both post operations are required to be active at the same time, and *asynchronous* otherwise. In both case, no information is saved and, therefore $U = \mathbf{1}$. Actually, drains are used to enforce synchronisation in the flow of data. Formally, an *asynchronous* drain is given by coalgebra

$$\llbracket \bullet \overset{\triangledown}{\vdash\!\!\relbar\!\!\relbar\!\!\dashv} \bullet \rrbracket \;:\; \mathbf{1} \longrightarrow \mathbf{1}^{\mathbb{D}+\mathbb{D}}$$

where both post ports are modelled by the (universal) function to $\mathbf{1}$, *i.e.*

$$\mathsf{post}_1 \;=\, !_{U\times\mathbb{D}} \;=\; \mathsf{post}_2$$

The same operations can be composed in a product to model the *synchronous* variant:

$$\llbracket \bullet \vdash\!\!\relbar\!\!\relbar\!\!\dashv \bullet \rrbracket \;:\; U \longrightarrow U^{\mathbb{D}\times\mathbb{D}}$$

defined by

$$\mathbf{1}\times(\mathbb{D}\times\mathbb{D}) \xrightarrow{\cong} \mathbf{1}\times\mathbb{D}\times\mathbf{1}\times\mathbb{D} \xrightarrow{\overline{\mathsf{post}}_1\times\overline{\mathsf{post}}_2} \mathbf{1}\times\mathbf{1} \xrightarrow{!} \mathbf{1}$$

There is an important point to make here. Note that in this definition two post ports were aggregated by a product, instead of resorting to the more common additive context. Such is required to enforce their simultaneous activation and, therefore, to meet the expected synchrony constraint. This type of port aggregation also appears as a result of concurrent composition. In general, when presenting a connector's interface, we shall draw a distinction between *single* and *composite* ports, the latter corresponding to the simultaneous activation of two or more of the former.

Composite ports, on the other hand, entail the need for a slight generalisation of hook. In particular it should cater for the possibility of a post port requiring, say, two values of type $\mathbb{D}$ be plugged to two (different) read or get ports. Such a generalisation is straightforward and omitted here (but used below on examples involving *drains*).

3.4.3 The Dining Philosophers

Originally posed and solved by Dijkstra in 1965, the *dinning philosophers* problem provides a good example to experiment an exogenous coordination model of the kind proposed in this thesis.

The basic version reads as follows. Five philosophers are seated around a table. Each philosopher has a plate of spaghetti and needs two forks to eat it. When a philosopher gets hungry, he tries to acquire his left and right fork, one at a time, in either order. If successful in acquiring two forks, he eats for a while, then puts down the forks and continues to think. In the sequel we discuss two possible solutions to this problem.

A *merger-drain* solution. One possible solution assumes the existence of five replicas of a component *Phi*(losopher), each one with four get ports, two on the lefthand side and another two on the righthand side. The port labeled left_i is activated by Phi_i to place a request for the left fork. On the other hand, port leftf_i is activated on its release (and similarly for the ports on the right). Coordination between them is achieved by a software connector Fork with four post ports, to be detailed below. The connection between two adjacent philosophers through a Fork is depicted in Fig. 3.4 which corresponds to the following expression in the calculus

$$(Phi_i \boxdot \mathsf{Fork}_i \boxdot Phi_{i+1}) \Lsh^{\mathsf{rr}_i\ \mathsf{rf}_i\ \mathsf{lr}_i\ \mathsf{lf}_i}_{\mathsf{right}_i\ \mathsf{rightf}_i\ \mathsf{left}_{i+1}\ \mathsf{leftf}_{i+1}} \tag{3.20}$$

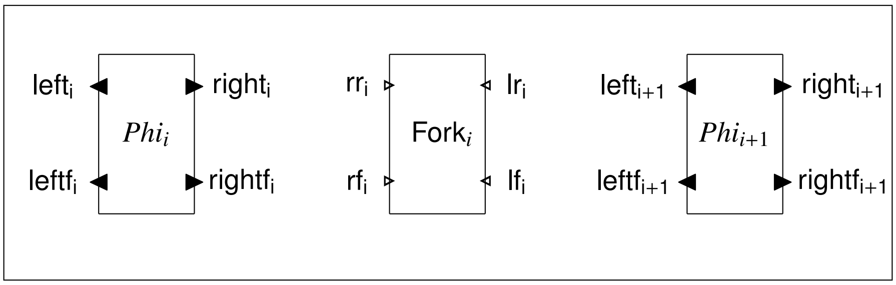


Figure 3.3: Dining Philosophers (1).

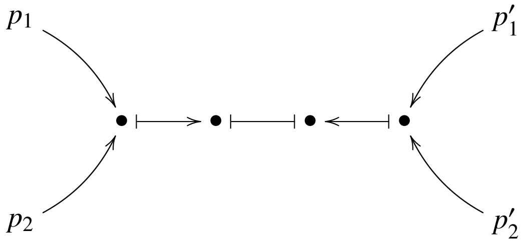


Figure 3.4: A *Fork* connector (1).

The synchronization constraints of the problem are dealt by connector Fork built from two blocking mergers and a synchronous drain depicted in figure 3.4 and given by expression

$$(\blacktriangleright \boxdot \blacktriangleright)\ ;\ \bullet\!\vdash\!\!\!\relbar\!\!\!\relbar\!\!\!\dashv\!\bullet \tag{3.21}$$

A *token* solution. Another solution is based on a specification of Fork as an *exchange token* connector. Such a connector is given as a coalgebra over $U = \{\pitchfork\} + \mathbf{1}$, where $\pitchfork$ is a token representing the (physical) fork. From the point of view of a philosopher requesting a fork equivales to an attempt to remove $\pitchfork$ from the connector state space. Dually, a fork is released by returning it to the connector state space. In detail, a fork request at a philosopher port, say right, which is a post port hooked to (the get port) rr of the connector is only successful if the token is available. Otherwise the philosopher must wait until a fork is released. The port specifications for Fork are as follows

$$\begin{aligned}\overline{\mathsf{rr}} = \overline{\mathsf{lr}}\ &:\ U \ \to\ U \times (\mathbb{D} + 1)\\ &=\ (=_{\pitchfork} \to\ (\iota_2 \cdot \underline{*}) \times (\iota_1 \cdot \underline{\pitchfork}),\ \mathsf{id} \times (\iota_2 \cdot \underline{*}))\end{aligned}$$

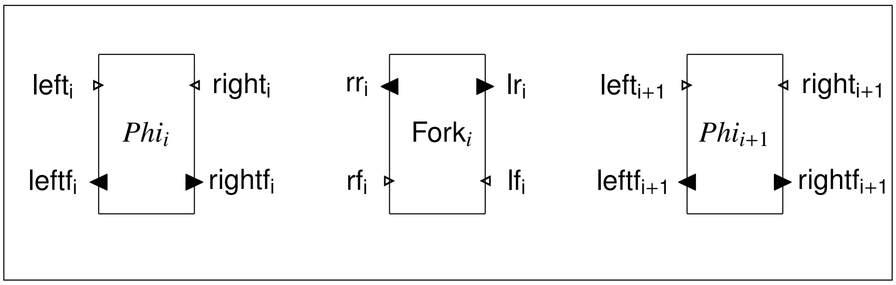


Figure 3.5: Dining Philosophers (2).

$$\begin{aligned}\overline{\mathsf{rf}} = \overline{\mathsf{lf}} \;:\; & U \times \mathbb{D} \;\rightarrow\; U \\ = \; & \iota_1 \cdot \underline{\pitchfork}\end{aligned}$$

Again, the *Fork* connector is used as a mediating agent between any two philosophers as depict in figure 3.5. The corresponding expression is

$$(Phi_i \boxdot \mathsf{Fork}_i \boxdot Phi_{i+1}) \Lsh^{\mathsf{right}_i\ \mathsf{rf}_i\ \mathsf{left}_i\ \mathsf{lf}_i}_{\mathsf{rr}_i\ \mathsf{rightf}_i\ \mathsf{lr}_{i+1}\ \mathsf{leftf}_{i+1}} \tag{3.22}$$

3.5 Towards a connector's calculus

3.5.1 Bisimilarity

If, in the previous chapter, relational equality and inclusion were used to model connector equivalence and refinement, respectively, standard coalgebraic tools can play a similar role in this new model. In particular, connectors being regarded as coalgebras, their equivalence can be expressed in terms of *bisimilarity*.

Bisimilarity is an equivalence relation among transition systems which relate states which allow identical observations and able to maintain such a property along their evolution. The original notion, introduced by David Park [Par81], is fundamental to process algebra (from the semantics of Ccs [Mil89] onwards). Coalgebra theroy [Rut00] made it generic, *i.e.*, parametric on the functor which captures the coalgebra dynamics. Formally,

Definition 2 (Bisimulation) *Given two* F*-coalgebras* $\langle U, \alpha\rangle$ *and* $\langle V, \beta\rangle$*, for a* Set *endofunctor* F*, a* F*-bisimulation is a relation* $R \subseteq U \times V$ *which is* closed under α and β*:*

$$(x, y) \in R \;\;\Rightarrow\;\; (\alpha(x), \beta(y)) \in \mathsf{F}R. \tag{3.23}$$

for all $x \in X$ and $y \in Y$.

A popular alternative formulation requires the possibility of R to be extended to a coalgebra ρ such that projections π_1 and π_2 lift to F-morphisms. Reference [BOS08] discusses bisimulations in a more general, essentially relational setting building on a synergy with Reynolds' *relation on functions* constructor involved in the *abstraction theorem* on parametric polymorphism [Rey83]. In particular it is shown that R is a bisimulation iff inequality

$$\alpha \cdot R \ \subseteq \mathsf{F}R \cdot \beta \tag{3.24}$$

holds in the category Rel of sets and relations.

In (3.23) and (3.24), $\mathsf{F}R$ stands for the relational *lifting* of R via functor F. This sends relation $R \subseteq X \times Y$ to relation $\mathsf{F}R \subseteq \mathsf{F}X \times \mathsf{F}Y$ is defined by induction on the structure of the functor F as follows

1. if F is the identity functor, then
$$\mathsf{F}\,R \ = \ R$$
2. if F is constant functor K, then
$$\mathsf{F}\,R \ = \ \mathsf{Id}_K$$
3. if F is a product $\mathsf{F}_1 \times \mathsf{F}_2$, then
$$(\mathsf{F}_1 \times \mathsf{F}_2)\,R \ = \ \{((x_1, y_1), (x_2, y_2)) \mid (x_1, x_2) \in \ \mathsf{F}_1 R \ \wedge (y_1, y_2) \ \in \ \mathsf{F}_2 R\}$$
4. if F is a coproduct $\mathsf{F}_1 + \mathsf{F}_2$, then
$$\begin{aligned}(\mathsf{F}_1 + \mathsf{F}_2)\,R \ = \ & \{((\iota_1, x_1), (\iota_1, x_2)) \mid (x_1, x_2) \in \ \mathsf{F}_1 R \wedge (y_1, y_2) \in \ \mathsf{F}_2 R\} \ \cup \\ & \{((\iota_2, y_1), (\iota_2, y_2)) \mid (y_1, y_2) \in \ \mathsf{F}_1 R \wedge (y_1, y_2) \in \ \mathsf{F}_2 R\}\end{aligned}$$
5. if F is an exponent G^A then
$$\mathsf{F}^A\ R \ = \ \{(f, g) \mid \forall_{a \in A} \cdot (fa, ga) \in \mathsf{F}R\}$$

Bisimilarity for functor F defined in (3.15) becomes the basis for establishing connector equivalence in this model. Often, however, when the signature of the connectors under discussion is fixed, it becomes simpler to derive a specific bisimulation scheme and express it directly in terms of the port operations. This is illustrated in the following two examples.

Example 3.1 *Let us consider, a basic connector c composed by a* post *and a* read *ports, the definition is given by the following coalgebra*

$$c = \langle \mathsf{post}, \mathsf{read} \rangle : X \longrightarrow X^{\mathbb{D}} \times (\mathbb{D} + 1)$$

For functor $\mathsf{F}X \;=\; \mathsf{id}^{\mathbb{D}} \times (\mathbb{D} + 1)$,

$$\mathsf{F}R \;=\; \{((f, \sigma), (g, \sigma')) \mid \forall_{d \,\in\, \mathbb{D}} \cdot \mathsf{F}R(f(d), g(d)) \;\wedge\; \sigma \;=\; \sigma'\}$$

Thus, a relation $R \subseteq X \times X$ is a bisimulation w.r.t. the connector signature functor if, for all $x, y \in X$,

$$\begin{aligned} & R(x, y) \Rightarrow \;\; FR\; ((\mathsf{post}(x), \mathsf{read}(x)), (\mathsf{post}(y), \mathsf{read}(y))) \\ \equiv & \\ & R(x, y) \Rightarrow \;\; \forall_{d \,\in\, \mathbb{D}} \cdot (\mathsf{post}(x)(d))\; R\; (\mathsf{post}(y)(d)) \;\wedge\; \mathsf{read}(x) \;=\; \mathsf{read}(y) \end{aligned}$$

Example 3.2 *Replacing a* read *port by a* get *port we have a connector where the state space U is updated both at input and output.*

$$c \;=\; \langle u \in U, \gamma_c : U \to (U \times (\mathbb{D} + 1))^{\mathbb{D}+\mathbf{1}} \rangle$$

Thus, calculating $\mathsf{F}R$*, for the functor* $\mathsf{F}X \;=\; (X \times (\mathbb{D} + 1))^{(\mathbb{D}+1)}$*, yields*

$$\mathsf{F}R \;= \{(f, g) | \;\langle \forall\; x \;:\; x \in \mathbb{D} + 1 \;:\; (\pi_1 f x, \pi_2 g x) \in R \;\wedge\; \pi_1 f x = \pi_2 g x \rangle\} \qquad (3.25)$$

which expresses that outputs observed through get *are identical and the relation is maintained for all pairs of states reached by either* post *or* get. *If* get *is expressed as a split of an attribute and a state transformer,* i.e.,

$$\mathsf{get} \;=\; \langle \mathsf{up}, \mathsf{ob} \rangle : X \longrightarrow X \times (\mathbb{D} + 1)$$

the bisimulation condition can be given in terms of the port operations considered. That is, relation $R \subseteq X \times X$ is a bisimulation if, for all $x, y \in X$,

$$\begin{aligned} & R(x, y) \Rightarrow \\ & \quad \langle \forall\; i \;:\; i \in (\mathbb{D} + 1) \;:\; \mathsf{post}(x)(d)\, R\, \mathsf{post}(y)(d) \;\wedge\; \mathsf{up}(u)\, R\, \mathsf{up}(u') \;\wedge\; \mathsf{ob}(x) = \mathsf{ob}(y) \rangle \end{aligned}$$

3.5.2 Reasoning about connectors

This sub-section illustrates the use of bisimilarity to establish, or disproof, connectors equivalence. Our small case study involves a discussion of the following law, which turns out to be false:

$$\mathsf{fifo}_1 \ ; \ \mathsf{fifo}_1 \ \sim \ \mathsf{fifo}_2 \tag{3.26}$$

Intuitively, fifo_1 is a one-place buffer, a basic connector in Reo, where it arises as the source of asynchrony in the corresponding coordination language [Arb04]. We start by defining fifo_1, and its two-place variant fifo_2, as coalgebras. Both have a post and a get port. The fact that coalgebras (in Set) are total functions, force us to introduce a special *empty* state which can be sent as an output and propagated. Formally, for the specification of fifo_1, let $U = \mathbb{D} + \mathbf{1}$ and define

$$\begin{aligned} \mathsf{post} : U &\longrightarrow U^{\mathbb{D}} \\ \mathsf{post}\,(\iota_1 u)\,(d) &= \iota_1 u \\ \mathsf{post}\,(\iota_2 *)\,(d) &= d \end{aligned}$$

and

$$\begin{aligned} \mathsf{get} : U &\longrightarrow U \times (\mathbb{D} + \mathbf{1}) \\ \mathsf{get}\, u &= (\iota_2 *, u) \end{aligned}$$

A fifo_2 connector is defined over $V = (\mathbb{D} + \mathbf{1}) \times (\mathbb{D} + \mathbf{1})$ with

$$\begin{aligned} \mathsf{post} : V &\longrightarrow V^{\mathbb{D}} \\ \mathsf{post}\,(\iota_2 *, \iota_2 *)\,(d) &= (\iota_2 *, d) \\ \mathsf{post}\,(\iota_2 *, \iota_1 x)\,(d) &= (d, \iota_1 x) \\ \mathsf{post}\,(\iota_1 x, v)\,(d) &= (\iota_1 x, v) \end{aligned}$$

and

$$\begin{aligned} \mathsf{get} : V &\longrightarrow V \times (\mathbb{D} + \mathbf{1}) \\ \mathsf{get}\,(v, v') &= ((\iota_2 *, v), v') \end{aligned}$$

Note that in both cases, the port specifications can be put together as coalgebra

$$\begin{aligned} \gamma : S &\longrightarrow (S \times (\mathbb{D} + \mathbf{1}))^{\mathbb{D}+\mathbf{1}} \\ \gamma\,(s, \iota_1 d) &= \mathsf{post}(s)\,(d) \\ \gamma\,(s, \iota_2 *) &= \mathsf{get}\, s \end{aligned}$$

where S is either U, for the fifo_1 connector, or V, for fifo_2.

Now let γ_1 and γ_2 be the dynamics of two fifo_1 connectors. Their interleaving $\gamma_1 \boxplus \gamma_2$ allows for the independent activation of each of its four ports over the product state space $U \times U$. In detail, one has post_1 and get_1 corresponding to the ports of the first fifo_1 operand, and correspondingly post_2 and get_2 for the second:

$$\begin{array}{rcl} \mathsf{post}_1\,(\iota_1 u, u')\,(d) & = & (\iota_1 u, u') \\ \mathsf{post}_1\,(\iota_2 *, u')\,(d) & = & (d, u') \\ \mathsf{get}_1\,(u, u') & = & ((\iota_2 *, u'), u) \end{array}$$

and

$$\begin{array}{rcl} \mathsf{post}_2\,(u, \iota_1 u')\,(d) & = & (u, \iota_1 u') \\ \mathsf{post}_2\,(u, \iota_2 *)\,(d) & = & (u, d) \\ \mathsf{get}_2\,(u, u') & = & ((u, \iota_2 *), u') \end{array}$$

The feedback of get_1 to post_2 in

$$\mathsf{fifo}_1 \ ; \ \mathsf{fifo}_1 \ \triangleq \ (\mathsf{fifo}_1 \boxplus \mathsf{fifo}_1) \, \hookleftarrow^{\mathsf{post}_2}_{\mathsf{get}_1} \tag{3.27}$$

eliminates those ports from the connectors interface and results and redefines the operations associated to the remaining post and get ports (actually, post_1 and get_2) as follows:

$$\begin{array}{rcl} \mathsf{post}\,(u, v)\,(d) & = & \mathsf{let}\ ((u', v'), x) = \mathsf{get}_1(u, v) \\ & & \quad \mathsf{if}\ x = \iota_1 d' \\ & & \quad\ \ \mathsf{then\ let}\ (u'', v'') = \mathsf{post}_2(u', v')\,(d')\ \mathsf{in}\ \mathsf{post}_1\,(u'', v'')\,(d) \\ & & \quad\ \ \mathsf{else}\ \mathsf{post}_1\,(u', v')\,(d) \end{array}$$

and

$$\begin{array}{rcl} \mathsf{get}\,(u, v) & = & \mathsf{let}\ ((u', v'), x) = \mathsf{get}_1(u, v) \\ & & \quad \mathsf{if}\ x = \iota_1 d' \\ & & \quad\ \ \mathsf{then\ let}\ (u'', v'') = \mathsf{post}_2(u', v')\,(d')\ \mathsf{in}\ \mathsf{get}_2\,(u'', v'') \\ & & \quad\ \ \mathsf{else}\ \mathsf{get}_2\,(u', v') \end{array}$$

Note that both definitions follow the same pattern: before executing the operation a sequence get_1 eventually followed by post_2 (if the output of get_1 is different from $\iota_2 *$), is performed, thus invisibly changing the connectors state space. This pattern is just a *pointwise* transliteration of function θ used in the definition of the *hook* combinator in section 3.3.

The *hook* definition entails the execution of, at least, the *output* operation involved which may result in some intuitively unexpected consequences. In this case this means that, in a situation in which both fifo_1 buffers are full performing a post or get operation will empty the first buffer as a first effect, leading to unwanted data lost. This would not be critical if the second buffer was empty, as in that case the removed data would be correctly transferred to it.

This means that connectors $\mathsf{fifo}_1\ ;\ \mathsf{fifo}_1$ and fifo_2 are not bisimilar, thus falsifying equation (3.26). Actually connector fifo_2 has a sort of *lookahead* capacity built in in its dynamics, which prevents a new data insertion, through port post if the two positions in the state space are already taken. Such a capacity is absent in composition $\mathsf{fifo}_1\ ;\ \mathsf{fifo}_1$: function θ forces the execution of the feedback sequence, or at least of part of it, irrespective of the state of the second fifo_1 connector.

Consider, now, the specification of the unbounded buffer connector fifo_∞ given in section 3.1 as a coalgebra over state space $U = \mathbb{D}^*$. Renaming its ports according to our general port designation scheme and translating the original definition into pointwise notation, yields

$$\begin{aligned} \mathsf{post}_\infty\,(u,d) &= \mathsf{receive}\,(u,d) \;=\; \mathsf{cons}(d,u) \\ \mathsf{get}_\infty\, u &= \mathsf{deliver}\, u \;=\; (\mathsf{tl}\, u, \mathsf{hd}\, u) \end{aligned}$$

Repeating the construction made above for the hook over the interleaving of two buffers, but now for the case

$$\mathsf{fifo}_1\ ;\ \mathsf{fifo}_\infty \;\triangleq\; (\mathsf{fifo}_1 \boxplus \mathsf{fifo}_\infty)\ \urcorner^{\mathsf{post}_2}_{\mathsf{get}_1} \tag{3.28}$$

results in the following definitions over the product of state spaces $U = (\mathbb{D}+\mathbf{1})\times\mathbb{D}^*$. Note that port p_∞ refers to p_∞ acting over the compound state space. Similarly for p_1 which is the p of the fifo_1 connector acting over $(\mathbb{D}+\mathbf{1})\times\mathbb{D}^*$.

$$\begin{aligned} \mathsf{post}\,(u,v)\,(d) \;&=\; \mathsf{let}\ ((u',v'),x) = \mathsf{get}_1(u,v) \\ &\qquad\quad \mathsf{if}\ x = \iota_1 d' \\ &\qquad\quad\ \ \mathsf{then\ let}\ (u'',v'') = \mathsf{post}_2(u',v')\,(d') \\ &\qquad\qquad\qquad \mathsf{in\ post}_1\,(u'',v'')\,(d) \\ &\qquad\quad\ \ \mathsf{else\ post}_1\,(u',v')\,(d) \\ = \{&\text{by definition of } \mathsf{post}_\infty\} \\ \mathsf{post}\,(u,v)\,(d) \;&=\; \mathsf{let}\ ((u',v'),x) = \mathsf{get}_1(u,v) \end{aligned}$$

$$\begin{aligned}
&\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{if}\ x = \iota_1 d' \\
&\quad\quad\quad\quad\quad\quad\quad\quad\ \ \mathsf{then\ post}_1\,(u', \mathsf{cons}(d', v'))\,(d) \\
&\quad\quad\quad\quad\quad\quad\quad\quad\ \ \mathsf{else\ post}_1\,(u', v')\,(d)
\end{aligned}$$

$= \{$by definition of get_1 and $\mathsf{post}_1\}$

$$\begin{aligned}
\mathsf{post}\,(u, v)\,(d) \ &= \ \mathsf{let}\ ((\iota_2 *, v), x) = \mathsf{get}_1(u, v) \\
&\quad\ \ \mathsf{if}\ x = \iota_1 d' \ \ \mathsf{then}\ (d, \mathsf{cons}(v, d'))\ \ \mathsf{else}\ (d, v)
\end{aligned}$$

and

$$\begin{aligned}
\mathsf{get}\,(u, v) \ &= \ \mathsf{let}\ ((u', v'), x) = \mathsf{get}_1(u, v) \\
&\quad\ \ \mathsf{if}\ x = \iota_1 d' \\
&\quad\ \ \ \ \mathsf{then\ let}\ (u'', v'') = \mathsf{post}_2(u', v')\,(d')\ \mathsf{in\ get}_1\,(u'', v'') \\
&\quad\ \ \ \ \mathsf{else\ get}_1\,(u', v')
\end{aligned}$$

$= \{$by definition of $\mathsf{post}_\infty\}$

$$\begin{aligned}
\mathsf{get}\,(u, v) \ &= \ \mathsf{let}\ ((u', v'), x) = \mathsf{get}_1(u, v) \\
&\quad\ \ \mathsf{if}\ x = \iota_1 d' \\
&\quad\ \ \ \ \mathsf{then\ get}_2\,((u', \mathsf{cons}(d', v')), d) \\
&\quad\ \ \ \ \mathsf{else\ get}_2\,((u', v'), d)
\end{aligned}$$

$= \{$by definition of get_1 and $\mathsf{get}_\infty\}$

$$\begin{aligned}
\mathsf{get}\,(u, v) \ &= \ \mathsf{let}\ ((\iota_2 *, v), x) = \mathsf{get}_1(u, v) \\
&\quad\ \ \mathsf{if}\ x = \iota_1 d' \ \ \mathsf{then}\ ((\iota_2 *, \mathsf{cons}(d', v)), \mathsf{hd\,cons}(d', v)) \\
&\quad\ \ \quad\quad\quad\quad\ \ \mathsf{else}\ ((x, \mathsf{tl}\,v), \mathsf{hd}\,v)
\end{aligned}$$

$= \{\mathsf{hd} \cdot \mathsf{cons} = \pi_1\}$

$$\begin{aligned}
\mathsf{get}\,(u, v) \ &= \ \mathsf{let}\ ((\iota_2 *, v), x) = \mathsf{get}_1(u, v) \\
&\quad\ \ \mathsf{if}\ x = \iota_1 d' \ \ \mathsf{then}\ ((\iota_2 *, \mathsf{cons}(d', v)), d') \\
&\quad\ \ \quad\quad\quad\quad\ \ \mathsf{else}\ ((x, \mathsf{tl}\,v), \mathsf{hd}\,v)
\end{aligned}$$

Comparing these definitions with those of the fifo_2 connector above, it is easy to exhibit a relation $R \subseteq (\mathbb{D} + \mathbf{1}) \times \mathbb{D}^*$ satisfying the bisimulation condition derived in *Example 2* in the previous sub-section. Alternatively one may show that isomorphism

$$\eta \ \triangleq \ (\mathbb{D} + \mathbf{1}) \times \mathbb{D}^* \xrightarrow{\mathsf{dr}} \mathbb{D} \times \mathbb{D}^* + \mathbb{D}^* \xrightarrow{[\mathsf{cons},\mathsf{id}]} \mathbb{D}^* \tag{3.29}$$

which in pointwise notation is given by the following two clauses:

$$\begin{aligned}
\eta\,(\iota_1 d, l) \ &= \mathsf{cons}(d, l) \\
\eta\,(\iota_2 *, l) \ &= l
\end{aligned}$$

is a morphism for coalgebras over $\mathsf{F}\,X \;=\; (X \times (\mathbb{D} + \mathbf{1}))^{(\mathbb{D}+\mathbf{1})}$, *i.e.*, such that the following diagram commutes:

$$\begin{tikzcd}
(\mathbb{D} + \mathbf{1}) \times \mathbb{D}^* \arrow[r, "\gamma_{(\mathsf{fifo}_1 \boxplus \mathsf{fifo}_\infty) \Lsh^{\mathsf{post}_2}_{\mathsf{get}_1}}"] \arrow[d, "\eta"'] & \mathsf{F}\,((\mathbb{D} + \mathbf{1}) \times \mathbb{D}^*) \arrow[d, "\mathsf{F}\,\eta"] \\
\mathbb{D}^* \arrow[r, "\gamma_{\mathsf{fifo}_\infty}"'] & \mathsf{F}\,\mathbb{D}^*
\end{tikzcd}$$

This shows that

$$\mathsf{fifo}_1 \;;\; \mathsf{fifo}_\infty \;\sim\; \mathsf{fifo}_\infty \tag{3.30}$$

Other laws of a connector calculus can be proved in a similar way. For example commutativity of both $\boxplus$ and $\boxdot$, *i.e.*,

$$c_1 \boxplus c_2 \;\sim\; c_2 \boxplus c_1 \tag{3.31}$$

$$c_1 \boxdot c_2 \;\sim\; c_2 \boxdot c_1 \tag{3.32}$$

can be proved by showing isomorphim $\mathsf{swap} : X \times Y \longrightarrow Y \times X$ is a coalgebra morphism for the functor capturing the signature of connectors c_1 and c_2. On the other hand, refinement results can be proved in this model through the construction of suitable *simulations*. The reader is referred to [MB05] for a generic (*i.e.*, parametric on the functor) way of computing simulations in a coalgebraic setting.

3.6 Towards mobile connectors

The increasing demand for complex and ubiquitous applications places new challenges to the way software is designed and developed. One of such challenges concerns the way in which an application deals with the dynamic reconfiguration of its components. Actually, components are no more static pieces of code assembled at compile time, but dynamic entities, often executing in different processing units, which interact only through well defined public interfaces. Component assembly is understood as interconnection of ports, declared in such interfaces, and more often than not such interconnection patterns change at runtime. This explains why, since the 1980's, *mobility* has become a buzzword both in academia and industry.

From a foundational point of view, the publication of Milner, Parrow and Walker original reports on the π-calculus [MPW92], in 1992, was a fundamental milestone. Since then the topic became increasingly popular in both theorectical and applied research. It arises, for example, in connection with software architecture (*e.g.*, [BGR$^+$99, Oqu04, NA03, GS04]), coordination models (*e.g.*, [Lum99, LF02]) or programming languages (*e.g.*, [Kir02, BTL05]), just to name a few. But still, in the practice of software engineering, mobility remains hard to express and to be reasoned about.

Mobility is structurally associated with distribution, system's temporal evolution and dynamic creation or reconfiguration of processes, links or components' instances. The model introduced in this chapter, however, assumes a fixed interconnection structure between the components and connectors involved, making the envisaged approach *static*. The remaining of this chapter reports preliminary work on a possible extension of this coalgebraic model to deal with dynamic reconfiguration. More precisely the extension consists of the explicit inclusion of a special connector — called the *orchestrator* — which plays the role of a connections manager. Such is achieved through a set of basic primitives to break or rebuild links between component's instances and connectors at run-time.

The proposed solution, which appeared in [BB07], is essentially *operational*: it does not contribute for explaining the mathematics of mobility, but provides a posible way of dealing with dynamic reconfiguration within an exogenous coordination model. The basic requirement placed by the exogenous nature of the model is the absence of direct communication between component's instances. One of them, for example, may decide at some point to disconnect from a connector's port and, let us suppose, to send that port identifier to be part of a new connection. All it can do, however, is to post the port identifier through another connector's port. What will happen afterwords, and in particular, which other component's instance, if any, will re-use such a port is not controlled by, in fact not even known to, the original component. In our approach each component's instance interfaces with the *glue* code, represented as a connector, and is not aware of the presence of other components equally connected to the same connector. That is why communication is anonymous and direct reconfiguration orders (like 'link this to that') are not possible.

Before detailing the specification of the *orchestrator* connector given in sub--section 3.6.2, we introduce a new element in the model to capture the *interface behaviour* of both connectors and components. This adds up to the static semantics discussed in the previous sections and it turns out to be a fundamental issue if one wants to deal with dynamic reconfiguration within the model. Such a notion of behavioural specification will be revisited and further elaborated in the model proposed in chapter 5. The version discussed in the sequel is enough for the moment.

3.6.1 Behaviour and configurations

In an exogenous coordination model component instances are always regarded as *black boxes*. All that is assumed to be known about them are

- the port interface signature, *i.e.*, the identifier and polarity of each of its ports
- a specification of the *interface behaviour*, which defines what is called here the *component's usage*, and denoted by *use*( ).

This is given by a process algebra-like expression and intended to define the activation pattern of the component interface. For example,

$$use(C) \;=\; (in.in.out)^*$$

establishes that the port activation pattern for component instance C requires two activations of port *in* before an activation of port *out*. The notation used is based on Ccs [Mil89] over a set Act of actions, each action corresponding to a port activation. Differently from Ccs, however, actions come equipped with a *co-occurence* operator $\|$ — action $a\|b$ stands for the simultaneous ocurrence of both a and b. Syntax is as follows:

$$B \;::=\; \mathbf{0} \mid a.B \mid B + B \mid B|B \mid B\backslash K \mid \sigma B \mid B^*$$

where $a \in Act$, $\cdot$ is the *prefix* operator, + and | denote *non deterministic choice* and *parallel* composition, respectively. Notation $B\backslash K$ represents *restriction* to a set K of actions, $[\sigma]$ stands for action *renaming* in accordance with substituion σ, and $*$ denotes *iteration*.

As described above, a connector is *internally* specified as a coalgebra built by port aggregation. Its external behaviour, however, is also given by a process algebra expression. For example, behaviour of a synchronous channel with port *in* and *out* is given by $(in\|out)^*$ whereas the asynchronous case is specified by $(in.out)^*$.

Component instances never interact with each other directly, but always through the connector network. Actually they are not even aware of each other's presence. The whole system is described by a number of component instances and a connector built from elementary connectors using a connector's combinators described early in this chapter[2]. The joint behaviour of connectors and component instances in a particular setting is called a $configuration$. This describes, in particular, the actual connections between ports.

[2] The extension of their definitions to the *behavioural* dimension is discussed, in a more general setting, in chapter 5.

3.6.2 A connector for orchestration

Central to our approach is the presence of a particular coordination connector, called the *orchestrator*, whose role is to manage the interactions among all other elements in the architectural network. In a sense, the *orchestrator* corresponds to an intermediary layer between components and ordinary connectors, *i.e.*, it acts as a bridge among components and connectors.

The *orchestrator* is a listener permanently attentive to the flow of messages which do not contain data but port identifiers, instead. Such messages will be understood as an order for control transfer. Actually, the orchestrator is triggered whenever a message with a port identifier *arrives* at a port of a particular component instance. This means that interception is made on the execution of a read or get operation. At this point, it intercepts the message, and re-organizes the overall network connections according to some *reconfiguration script*. Notice, however, that neither component's instances nor the connectors in the network are aware of its presence.

To fulfill its role the *orchestrator* is defined as a coalgebra over state space U specified as datatype

$$U \;:\; \mathbb{P} \times Cmp \times \mathcal{P}(\mathbb{P} \times \mathbb{P}) \tag{3.33}$$

where $\mathbb{P}$ is the set of *port identifiers* known to the orchestrator, $\mathcal{P}(\mathbb{P} \times \mathbb{P})$ is a set of active *connections* and Cmp is a *component manager* defined as function

$$Cmp = (\mathcal{P}\mathbb{P})^{cmpId} \tag{3.34}$$

which associates to each component instance, identified by a value of type *cmpId*, the set of its ports. There is an obvious type invariant associated to U stating that all connections are point-to-point:

$$\mathsf{inv}\, u \;=\; \langle \forall\, p \;:\; p \in \pi_1 u :\; \mathsf{card}\,\{c \in \pi_3 u \,|\, \pi_1 c = p \,\vee\, \pi_2 c = p\} \;\leq\; 1 \rangle \tag{3.35}$$

The *Orchestrator* is equipped with a set of primitives to manage connections and allow their dynamic (re)-configuration. The underlying operations are defined as follows

- *Get Connection* (getCon): This operation takes a port identifier and, if such an identifier is part of a connection, returns the corresponding end point. Formally,

$$
\begin{aligned}
\mathsf{getCon} &: U \times \mathbb{P} \to \mathbb{P} + 1 \\
\mathsf{getCon}(u, p) &\triangleq \\
&\quad \mathsf{let} \\
&\quad\quad e \;=\; \{\pi_2 c |\; c \in \pi_3 u \;\wedge\; \pi_1 c = p\} \cup \{\pi_1 c |\; c \in \pi_3 u \;\wedge\; \pi_2 c = p\} \\
&\quad \mathsf{in} \\
&\quad\quad (e \neq \emptyset \;\to\; \iota_1\, \mathsf{the}(e),\; \iota_2\, \bot)
\end{aligned}
$$

- *Disconnection* (*disCon*): This operation updates the connector's state space by deleting an existing connection.

$$
\begin{aligned}
\mathsf{disCon} &: U \times (\mathbb{P} \times \mathbb{P}) \to U \\
\mathsf{disCon}(u, (p, p')) &\triangleq (\pi_1 u, \pi_2 u, (\pi_3 u) \backslash \{(p, p')\})
\end{aligned}
$$

- *Add Port* (*addPort*): This operation updates the connector's state space by adding a new available port.

$$
\begin{aligned}
\mathsf{addPort} &: U \times \mathbb{P} \to U \\
\mathsf{addPort}(u, p) &\triangleq (\{p\} \cup \pi_1 u, \pi_2 u, \pi_3 u)
\end{aligned}
$$

- *Available Ports (avPort)*: This operation searches for ports in a given component instance available for connection.

$$
\begin{aligned}
\mathsf{avPort} &: U \times cmpId \to \mathcal{P}(\mathbb{P}) \\
\mathsf{avPort}(u, i) &\triangleq \\
&\quad \mathsf{let} \\
&\quad\quad \mathsf{used} \;=\; \mathcal{P}(\pi_1)\,(\pi_3 u) \;\cup\; \mathcal{P}(\pi_2)\,(\pi_3 u) \\
&\quad \mathsf{in} \\
&\quad\quad (\pi_3 u)\, i \;\cap\; \mathsf{used}
\end{aligned}
$$

- *Make Connection* (*mkCon*): This operation aggregates a new connection to the orchestrator's state space.

$$
\begin{aligned}
\mathsf{mkCon} &: U \times (\mathbb{P} \times \mathbb{P}) \to U \\
\mathsf{mkCon}(u, (p, p')) &\triangleq (\pi_1 u, \pi_2 u, \{(p, p')\} \cup (\pi_3 u))
\end{aligned}
$$

3.6.3 Coordination Patterns

The set of primitives specified above are used to build the already mentioned *reconfiguration scripts* which constitute the orchestrator reaction to the interception of (incoming) messages. The idea is that the orchestrator's behaviour is parameterized by such scripts, which, given their role in the model, will be also referred to as *coordination pattern*. Let us consider one of such patterns to illustrate how they can be specified in terms of the orchestrator primitives.

The intuition on this patterns is as follows: On interception of a message containing a port identifier m arriving to port p, the orchestrator identifies the component instance to which p belongs and tries to find another port in it to connect to m. In case m was previously part of another connection, such connection has to be traced and broken. Formally,

$$\begin{aligned}
\mathsf{pattern}_1(u,p,m) \;\triangleq\; & \\
& \mathsf{let} \\
& \quad ap \;=\; \mathsf{avPort}(u, \mathsf{owner}(u,p)) \\
& \mathsf{in} \\
& \quad (ap \neq \emptyset \;\rightarrow\; let\ r \in ap\ in\ \mathsf{reconf}(u,p,m,r)\,,\ u)
\end{aligned}$$

where

$$\begin{aligned}
\mathsf{reconf}(u,p,m,r) \;\triangleq\; & \\
& \mathsf{let} \\
& \quad x \;=\; \mathsf{getCon}(u,m) \\
& \mathsf{in} \\
& \quad (x \;=\; \iota_1(m') \;\rightarrow\; \mathsf{mkCon}(\mathsf{disCon}(u,(m,m')),(m,r))\,,\ \mathsf{mkCon}(u,(m,r)))
\end{aligned}$$

and function owner inspects the *component manager* in the orchestrator's state to return the identifier of the component instance to which a particular port belongs. Formally,

$$\mathsf{owner}(u,p) \;\triangleq\; \mathsf{the}\,\{c \in cmpId|\ p \in (\pi_2 u)\,c\}$$

Note that in this coordination pattern if there is no port available for the new connection, the configuration if not changed. This is not, however, the only possibility. Reasonable alternatives would be

- to disconnect port m in any case
- or to suspend until the a port becomes available for connection in owner(u,p).

Such alternatives can also be encoded as *coordinating patterns* to act as a parameter to the orchestrator.

3.6.4 An example

For illustration purposes let us consider how to model, in the framework outlined in the previous sections a variation of the example presented in [Mil99].

We consider a *wireless network* where a notebook (component *Client*) is connected to a network which has two servers ($Base_1$ and $Base_2$). These two servers are connected to each other and the client is connected to one of them.

In the initial configuration of the system the *Client* is communicating with the server $Base_1$ according to the Fig 3.6. The *Client* may permanently communicate with the network through its input port c_i and output port c_o. Such a behaviour is captured by

$$use(Client) \ = \ (c_i \ + \ c_o)^*$$

$Base_1$ is permanently communicating with $Base_2$, through its output port $b_{1.3}$ and input port $b_{1.4}$. In such a system configuration $Base_1$ is also communicating with the *Client* using the output port $b_{1.1}$ and the input port $b_{1.2}$. This behaviour is given by,

$$use(Base_1) \ = \ (b_{1.1} \ + \ b_{1.2} \ + \ b_{1.3} \ + \ b_{1.4})^*$$

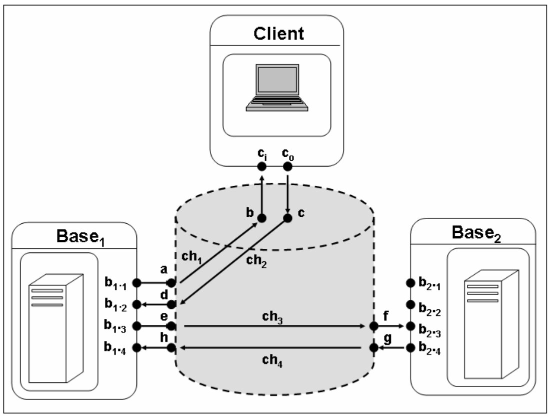


Figure 3.6: The initial configuration.

$Base_2$ communicates with $Base_1$ though its input port $b_{2.3}$ and output port $b_{2.4}$. $Base_2$ also has an input port $b_{2.1}$ and an output port $b_{2.2}$. In this case both ports are available and disconnected from the connectors network. The behaviour of $Base_2$ is given by,

$$use(Base_2) \;=\; (b_{2.3} \;+\; b_{2.4})^*$$

The components involved in the network are interconnected by a connector made of the four synchronous channels depicted in Fig 3.6. Its behaviour, $use(C)$, is obtained by the parallel composition of each channel. Behaviour composition will be further discussed in chapter 5. For the moment it is enough to point out that, as usual, the semantics of parallel composition is given in terms of both interleaving and synchronous product.

The whole system is specified by configuration $conf$ whose behaviour is

$$\begin{aligned} use(conf) \;=\; & use(Client)[b/c_i, c/c_o] \mid use(C) \\ & \mid\; use(Base_1)[a/b_{1.1},\; d/b_{1.2},\; e/b_{1.3},\; h/b_{1.4}] \\ & \mid\; use(Base_2)[f/b_{2.3},\; g/b_{2.4}] \end{aligned}$$

Note that the renaming operation connects the components ports to the connectors ports.

Now, let us consider that the user moves and the signal becomes weak. The server $Base_1$ communicates with the server $Base_2$ and sends the port identifiers so that $Base_2$ becomes in charge of providing the service. The Fig 3.7 represents the result of such an operation.

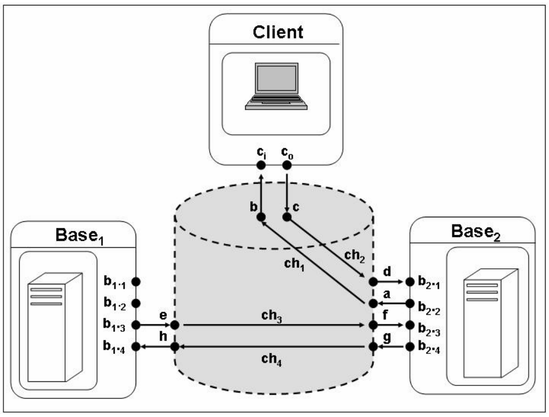


Figure 3.7: The final configuration.

After the *orchestrator* has intercepted the message the behaviour of the configuration becomes

$$\begin{aligned} use(conf') &= use(Client)[b/c_i, c/c_o] \mid use(C) \\ &\mid use(Base_1)[e/b_{1.3}, h/b_{1.4}] \\ &\mid use(Base_2)[d/b_{2.1}, a/b_{2.2}, f/b_{2.3}, g/b_{2.4}] \end{aligned}$$

Let us now focus on the *orchestrator* role. Suppose it is parametrized with $\mathsf{pattern}_1$ above and let its initial state be $u = \langle p, cmp, con \rangle$, where

$$\begin{aligned} p &= \{c_i, c_o, b_{1.1}, b_{1.2}, b_{1.3}, b_{1.4}, b_{2.1}, b_{2.2}, b_{2.3}, b_{2.4}\} \\ cmp &= \{(Client, \{c_i, c_o\}), (Base_1, \{b_{1.1}, b_{1.2}, b_{1.3}, b_{1.4}\}), (Base_2, \{b_{2.1}, b_{2.2}, b_{2.3}, b_{2.4}\})\} \\ con &= \{(a, b), (c, d), (e, f), (g, h)\} \end{aligned}$$

Suppose, now, the following situation occurs: $Base_1$ sends port identifier a of channel ch_1 to $Base_2$ through the connector C. The *orchestrator* captures such a message and starts the script defined in $\mathsf{pattern}_1$, as follows.

- With avPort the *orchestrator* selects port $b_{2.2}$ as an alternative to connect to a.
- With operation getCon it obtains port $b_{1.1}$, which was previously connected to a.
- As such a port has a current connection, the following step is to break it with disCon.
- Finally, the new connection, linking a to $b_{2.2}$ is completed.

As this example shows, by designing suitable *patterns* to feed the orchestrator, a number of dynamic reconfiguration situations can be (at least operationally) handled within the model.

The idea underlying the behaviour specification, *use*( ), will be generalised in chapter 4, to specify components interfaces, an important ingredient of the model proposed in chapter 5.

Chapter 4

Behavioural Interfaces

Summary

State-based connectors were introduced, in the previous chapter, as an extension of elementary input-output relations to coalgebras. Making connectors context-aware, i.e., *able to sense their environment and actuate accordingly, entails a deeper update of our basic model. In particular it becomes necessary to endow interfaces, of both components and connectors, with a specification of their behaviour or interaction protocol. Process algebras are a natural candidate for such specifications. This, however, entails the need for more generic and adaptable approaches to their design. For example, similar combinators coexisting with different interaction disciplines may be required. In such a context, this chapter pursues a research programme on a coinductive rephrasal of classic process algebra documented in [Bar01, BO02]. A particular emphasis is put on the study of interruption combinators defined by natural co-recursion. The chapter also illustrates the verification of their properties in an pointfree reasoning style as well as their direct encoding in* HASKELL. *Finally, behaviour-annotated interfaces are defined paving the way to a model of context-aware connectors to be introduced in chapter 5. The idea of behavioural interfaces, in the context of our research, was first introduced in [BBC07]. The technical contributions of the chapter were published in [RBB06].*

4.1 Interfaces and behaviour

Usually, a system architecture is depicted by block diagrams representing components and arrows corresponding to their interconnections. In order to capture accurately the high-level structure of a system components have to make public the services they provide and how they should be used. In such a context *interfaces* play a fundamental role.

Along the two previous chapters interfaces were restricted to provide only pure syntactic information, *i.e.*, the names of services (each one associated to a specific port) and the types of their parameters. Formally interfaces were identified with algebraic signatures.

Often, however, this view is too restrictive. In particular, it fails to describe when services may be invoked and what sequences of interactions are allowed for each component. This is sometimes referred to as the *interaction protocol* or the *component behaviour*.

Enriching component interfaces with this sort of behavioural information turns out to be fundamental if the envisaged coordination layer is supposed to include *context-aware* connectors, *i.e.*, connectors able to sense their environment and actuate accordingly. Such is the justification for this chapter, previous to the introduction of a model for context awareness in chapter 5.

The idea of behaviour-annotated interfaces, in itself, is not new. On the contrary, it appears in most modern ADLs (Architectural Description Languages), where behaviour is expressed through transition systems [MKG99], regular-expressions [PV02] or process algebras [AG97]. Moreover, the formal treatment of interfaces (as in, *e.g.*, [AH01]), supports compositional design for checking both interface compatibility and refinement.

Process algebra [PS01], in particular, provides an expressive setting for representing behavioural patterns and establish or verify their properties in a compositional way. Each process algebra introduces a number of combinators for processes (or *behaviours*) and an interaction discipline, for example synchronisation of complementary labels in Ccs [Mil89], or of identical named actions in Csp [Hoa85]. In the context of component coordination, however, sticking to a fixed interaction discipline is a severe limitation. Actually, different such disciplines have to be used, at the same time, to capture different aspects of component coordination. As discussed in chapter 5, the discipline governing the composition of software connectors (to build the overall *glue code*) differs from the one used to capture the interaction between the connectors and the relevant components' interfaces. Thus some flexibility in the definition and support of interaction disciplines is required.

To meet this goal, which entails the need for a *generic* way to *design* process algebras, we built on top of previous work at Minho documented in references [Bar01, BO02]. Such an approach is revisited in the following section. Afterwards, sections 4.3 and 4.4 present our own contribution to this approach. In particular, a generic version to the expansion law is proved in section 4.3, which is heavily used in chapter 5. Section 4.4 introduces combinators for composition with *interruption* and *recovery*, a topic quite promising for extending the coordination models discussed in this thesis, but not used further in the thesis. Finally, in section 4.5, we come back to the thesis main stream, introducing behaviour-annotated interfaces for both components and connectors.

4.2 Revisiting process algebra

4.2.1 Introduction

This section provides an introduction to our own coalgebraic approach to the design of generic process algebra. References [Bar01, BO02] introduce a denotational approach to the *design* of process algebras in which processes are identified with inhabitants of a final coalgebra and their combinators defined by coinductive extension (of 'one-step' behaviour generator functions). The goal was to apply to this area of computing the same reasoning principles and calculational style one gets used to in the *dual* world of functional programming with algebraic datatypes. Actually, it is well known that *initial* algebras and *final* coalgebras provide abstract descriptions of *data* and *behavioural* structures, respectively. Both initiality and finality, as universal properties, entail definitional and proof principles, which form a basis for the development of program calculi directly based on (actually driven by) type specifications. The role of universals constructions, such as *initial* algebras and *final* coalgebras, is — when combined with the 'calculational' style entailed by category theory — fundamental to a whole discipline of algorithm derivation and transformation, which can be traced back to the so-called *Bird-Meertens formalism* [BM87] and the foundational work of T. Hagino [Hag87b]. Dually, our research programme regards *processes* as inhabitants of coinductive types, *i.e.*, final coalgebras for the powerset functor $\mathcal{P}(Act \times \mathsf{Id})$, where *Act* denotes a set of *action* identifiers. Finally, process *combinators* are defined as *anamorphisms* [MFP91], *i.e.*, by coinductive extension. Note that, if coalgebras for a functor T can be regarded as generalisations of transition systems of shape T, their behavioural patterns are revealed by the successive observations allowed by the signature of observers recorded in T. Then, just as initial algebras are canonnically defined over the terms generated by successive application of constructors, such 'pure' observed behaviours form the state spaces of final coalgebras. It comes with no surprise that *bisimulation* coincides with equality in such coalgebras. Therefore our approach has the attraction of replacing proofs by bisimulation, which as in *e.g.*, [Mil89], involves the explicit construction of such a relation, by *equational reasoning*. Recently this approach has been extended to capture weak equivalences, as documented in [RBW07].

Technically, our approach amounts to the systematic use of the universal property which characterizes *anamorphisms*. Recall that, for a functor T and an arbitrary coalgebra $\langle U, p : U \longrightarrow \mathsf{T}\, U\rangle$, an *anamorphism* is the *unique* morphism to the final coalgebra $\omega_{\mathsf{T}} : \nu_{\mathsf{T}} \longrightarrow \mathsf{T}\ \nu_{\mathsf{T}}$. Written, in the tradition of [MFP91], as $[\![(p)]\!]_{\mathsf{T}}$ or, simply, $[\![(p)]\!]$, an anamorphism satisfies the following universal property:

$$k = [\![(p)]\!]_{\mathsf{T}} \quad \leftrightarrow \quad \omega_{\mathsf{T}} \cdot k = \mathsf{T}\ k \cdot p \tag{4.1}$$

which corresponds to the commutativity of the following diagram

$$\begin{tikzcd}
\nu_{\mathsf{T}} \arrow[r, "\omega_{\mathsf{T}}"] & \mathsf{T}\ \nu_{\mathsf{T}} \\
U \arrow[u, "{[\![(p)]\!]_{\mathsf{T}}}"] \arrow[r, "p"] & \mathsf{T}\ U \arrow[u, "{\mathsf{T}\ [\![(p)]\!]_{\mathsf{T}}}"']
\end{tikzcd}$$

from which the following *cancellation*, *reflection* and*fusion* laws are easily derived:

$$\omega_{\mathsf{T}} \cdot [\![(p)]\!] \;=\; \mathsf{T}\ [\![(p)]\!] \cdot p \tag{4.2}$$

$$[\![(\omega_{\mathsf{T}})]\!] \;=\; \mathsf{id}_{\nu_{\mathsf{T}}} \tag{4.3}$$

$$[\![(p)]\!] \cdot h \;=\; [\![(q)]\!] \quad \text{if} \quad p \cdot h \;=\; \mathsf{T}\ h \cdot q \tag{4.4}$$

The *existence* assertion underlying (4.1) (corresponding to the left to right implicants) provides a *definition* principle for (circular) functions to the final coalgebra which amounts to equip their source with a coalgebraic structure specifying the *next-step* dynamics. We call such a coalgebra the *gene* of the definition: it carries the 'genetic inheritance' of the function. Then the anamorphism gives the rest. The *uniqueness* part, underlying right to left implication in (4.1), on the other hand, offers *coinduction* as a *proof* principle.

4.2.2 Combinators

In this approach, transition systems over a state space U and a set A of labels, classicaly specified as *binary relations*

$${}_{\alpha}\!\longleftarrow\, : U \longrightarrow A \times U \tag{4.5}$$

are given by coalgebras

$$\alpha : U \longrightarrow \mathcal{P}(A \times U) \tag{4.6}$$

for $\mathcal{P}(A \times \mathsf{Id})$, where $\mathcal{P}$ and Id denote, respectively, the (finite) powerset and the identity functor. It is well-known that set-valued functions, such as coalgebra (4.6) are models of binary relations and, conversely, any such relation is uniquely transposed into a set-valued function. The existence and uniqueness of such a transformation leads to the identification of a *transpose* operator Λ [BM97] characterized by an universal property which, for this particular case, reads

$$\alpha = \Lambda \; {}_{\alpha}\!\longleftarrow \quad \equiv \quad {}_{\alpha}\!\longleftarrow = \in \cdot\, \alpha \tag{4.7}$$

where ∈ denotes set membership and · is relational composition. Moreover, whenever $\mathcal{P}$ in (4.6) is restricted to the *finite* powerset, to enforce the existence of a final universe, equivalence (4.7) establishes again a bijective correspondence between the resulting coalgebras and *image finite* relations.

In [Bar01] processes are regarded as inhabitants of the final coalgebra

$$\omega : \nu \longrightarrow \mathcal{P}(Act \times \nu)$$

where $\mathcal{P}$ is the finite powerset functor. The restriction to the finite powerset avoids cardinality problems and assures the existence of a final coalgebra for T. This means, of course, we are restricted to *image-finite* processes,

The carrier of ω is the set of possibly infinite labelled trees, finitely branched and quotiented by the greatest bisimulation [Acz93], on top of which process combinators are defined. The first set contains the so-called *dynamic* combinators, *i.e.*, combinators which are 'consumed' on action occurrence, disappearing from the expression representing the process continuation. Typical examples (from *e.g.* [Mil89]) include *inaction*, *prefix* and non-deterministic *choice*. The first one is represented as a constant $\mathsf{nil} : \mathbf{1} \longrightarrow \nu$ upon which no relevant observation can be made. Prefix gives rise to an *Act*-indexed family of operators $a. : \nu \longrightarrow \nu$, with $a \in Act$. Finally, the possible actions of the non deterministic choice of two processes p and q corresponds to the collection of all actions allowed for p and q. Formally,

$$\begin{aligned} &\textit{inaction} && \omega \cdot \mathsf{nil} \;=\; \underline{\emptyset} && (4.8)\\ &\textit{prefix} && \omega \cdot a. \;=\; \mathsf{sing} \cdot \mathsf{label}_a && (4.9)\\ &\textit{choice} && \omega \cdot + \;=\; \cup \cdot (\omega \times \omega) && (4.10) \end{aligned}$$

where $\mathsf{sing} = \lambda x\,.\; \{x\}$ and $\mathsf{label}_a = \lambda x\,.\; \langle a, x\rangle$. Recursive combinators, on the other hand, are defined as anamorphisms. A typical example is *interleaving* $\boxplus : \nu \times \nu \longrightarrow \nu$ which represents an interaction-free form of parallel composition. The following definition captures the intuition that the observations over the interleaving of two processes correspond to all possible interleavings of observations of their arguments. Thus, $\boxplus \;=\; [\!(\alpha_{\boxplus})\!]$, where

$$\begin{aligned} \alpha_{\boxplus} \;=\; & \nu \times \nu \xrightarrow{\;\triangle\;} (\nu \times \nu) \times (\nu \times \nu) \xrightarrow{(\omega \times \mathsf{id}) \times (\mathsf{id} \times \omega)} (\mathcal{P}(Act \times \nu) \times \nu) \times (\nu \times \mathcal{P}(Act \times \nu)) \\ & \xrightarrow{\tau_r \times \tau_l} \mathcal{P}(Act \times (\nu \times \nu)) \times \mathcal{P}(Act \times (\nu \times \nu)) \xrightarrow{\;\cup\;} \mathcal{P}(Act \times (\nu \times \nu)) \end{aligned}$$

Morphisms $\tau_r : \mathcal{P}(Act \times X) \times C \longrightarrow \mathcal{P}(Act \times (X \times C))$ and $\tau_l : C \times \mathcal{P}(Act \times X) \longrightarrow \mathcal{P}(Act \times (C \times X))$ stand for, respectively, the right and left *strength* associated to functor $\mathcal{P}(Act \times \mathsf{Id})$.

4.2.3 Interaction

The *synchronous product* models the simultaneous execution of two processes, which, in each step, interact through the actions they realize. Let us, for the moment, represent such interaction by a function $\theta : Act \longrightarrow Act \times Act$. Formally, $\boxtimes = [\![(\alpha_{\boxtimes})]\!]$ where

$$\alpha_{\boxtimes} = \nu \times \nu \xrightarrow{(\omega \times \omega)} \mathcal{P}(Act \times \nu) \times \mathcal{P}(Act \times \nu) \xrightarrow{\mathsf{sel} \cdot \delta_r} \mathcal{P}(Act \times (\nu \times \nu))$$

where sel filters out all synchronisation failures. and δ_r is given by

$$\delta_r \langle c_1, c_2 \rangle = \{\langle a' \, \theta \, a, \langle p, p' \rangle \rangle | \, \langle a, p \rangle \in c_1 \, \wedge \, \langle a', p' \rangle \in c_2\}$$

But what is θ? This operation defined over *Act* what we call an *interaction structure*: *i.e.*, an Abelian positive monoid $\langle Act; \theta, 1 \rangle$ with a zero element 0. It is assumed that neither 0 nor 1 belong to the set of elementary actions. The intuition is that θ determines the interaction discipline whereas 0 represents the absence of interaction: for all $a \in Act, a\theta 0 = 0$. On the other hand, a positive monoid entails $a\theta a' = 1$ iff $a = a' = 1$. The role of 1, often regarded as an *idle* action, is essentially technical. Notice that the role of both 0 and 1 is essentially technical in the description of the interaction discipline. In some situations 1 may be seen as an *idle* action, but its role, in the general case, is to equip the behaviour functor with a monadic structure, which would not be the case if *Act* were defined simply as an Abelian semigroup. This structure was inpired by Winskel's *synchronisation algebras* [WN95], and is, in fact, the main source of *genericity* in our approach.

As a matter of fact by parameterizing a basic calculus by an interaction structure, one becomes able to design quite a number of different, application-oriented, process combinators. For example, Ccs assumes a set L of labels with an involutive operation, represented by an horizontal bar as in $\overline{a}$. Any two actions a and $\overline{a}$ are called complementary and a special action $\tau \notin L$ is introduced to represent the result of a synchronisation between a pair of complementary actions. Therefore, the result of θ is τ whenever applied to a pair of complementary actions and 0 in all other cases, except, obviously, if one of the arguments is 1. In Csp, on the other hand, $a\theta a = a$ for all action $a \in Act$. Yet other examples emerge in component coordination. Typically a *glass-box view* of a particular architectural configuration (*i.e.*, a 'glued' set of components and software connectors) will call for a *co-occurrence* interaction: θ is defined as $a\theta b = \langle a, b\rangle$, for all $a, b \in Act$ different from 0 and 1. For the *black-box view*, however, actions are taken as sets of labels, and θ defined as set intersection.

Synchronous product depends in a crucial way on the interaction structure adopted. For example its commutativity depends only on the commutativity of the underlying θ. Such is also the case of the standard *parallel composition* which combines the effects of both $\boxplus$ and $\boxtimes$. Note, however, that such a combination is performed at the *genes* level:

$$\boxdot \;=\; [\!(\alpha_{\boxdot})\!] \tag{4.11}$$

where

$$\begin{aligned} \alpha_{\boxdot} \;=\; & \nu \times \nu \xrightarrow{\;\triangle\;} (\nu \times \nu) \times (\nu \times \nu) \xrightarrow{(\alpha_{\boxplus} \times \alpha_{\boxtimes})} \\ & \mathcal{P}(Act \times (\nu \times \nu)) \times \mathcal{P}(Act \times (\nu \times \nu)) \xrightarrow{\;\cup\;} \mathcal{P}(Act \times (\nu \times \nu)) \end{aligned}$$

4.3 Proofs and prototypes

As mentioned above, definition and proof by coinduction forms the base of the Minho approach to process calculi design. In this section we

- illustrate the underlying rationale by considering the definition of a new combinator $\backslash_k$ whose aim is to make internal all occurrences of a specific action k;
- prove a generic version of Milner's expansion law;
- and introduces (a fragment of) a Haskell library for prototyping process algebras directly based on the coinductive definitions.

4.3.1 Hiding

Let us first consider the definition of a new combinator, $\backslash_k$, whose aim is to make internal all occurrences of a specific action k. Thus,

$$\backslash_k \;=\; [\!(\alpha_k)\!]$$

with

$$\alpha_k \;=\; \nu \xrightarrow{\omega} \mathcal{P}(Act \times \nu) \xrightarrow{\mathcal{P}(\mathsf{sub}_k \times \mathsf{id})} \mathcal{P}(Act \times \nu)$$

and $\mathsf{sub}_k \;\triangleq\; (=_k) \to \tau, \mathsf{id}$, τ standing for a representation of an *internal* action. Once defined a combinator, its theory arises by finding out how it interacts with the rest of the algebra. We consider now interaction with *interleaving*. This provides a first example of a coinductive proof by calculation, to be opposed to the more classic proof by bisimulation.

Lemma 4.1

$$\backslash_k \cdot \boxplus \;=\; \boxplus \cdot (\backslash_k \times \backslash_k) \tag{4.12}$$

Proof. Note that equation (4.12) does not allow a direct application of the fusion law. Since ω is an isomorphism, however, we may rewrite it as

$$\omega \cdot \backslash_k \cdot \boxplus = \omega \cdot \boxplus \cdot (\backslash_k \times \backslash_k) \tag{4.13}$$

which can be further simplified in terms of the corresponding genes, because both $\boxplus$ and $\backslash_k$ were defined by coinduction. Consider first the left hand side of (4.13).

$$\begin{aligned}
& \omega \cdot \backslash_k \cdot \boxplus \\
= & \qquad \{ \text{ definition of } \omega \cdot \backslash_k \text{, cancellation} \} \\
& \mathcal{P}(\mathsf{id} \times \backslash_k) \cdot \alpha_k \cdot \boxplus \\
= & \qquad \{ \text{ definition of } \alpha_k \} \\
& \mathcal{P}(\mathsf{id} \times \backslash_k) \cdot \mathcal{P}(\mathsf{sub}_k \times \mathsf{id}) \cdot \omega \cdot \boxplus \\
= & \qquad \{ \boxplus \text{ is a morphism } \} \\
& \mathcal{P}(\mathsf{id} \times \backslash_k) \cdot \mathcal{P}(\mathsf{sub}_k \times \mathsf{id}) \cdot \mathcal{P}(\mathsf{id} \times \boxplus) \cdot \alpha_{\boxplus} \\
= & \qquad \{ \text{ functors and definition of } \alpha_{\boxplus} \} \\
& \mathcal{P}(\mathsf{id} \times \backslash_k) \cdot \mathcal{P}(\mathsf{id} \times \boxplus) \cdot \mathcal{P}(\mathsf{sub}_k \times (\mathsf{id} \times \mathsf{id})) \cdot \cup \cdot (\tau_r \times \tau_l) \cdot (\omega \times \mathsf{id}) \times (\mathsf{id} \times \omega) \cdot \Delta \\
= & \qquad \{ \cup, \tau_r \text{ and } \tau_l \text{ are natural } i.e. \tau_r \cdot (\mathsf{B}f \times g) = \mathsf{B}(f \times g) \cdot \tau_r \text{ e } \tau_l \cdot (f \times \mathsf{B}g) = \mathsf{B}(f \times g) \cdot \tau_l \text{ for } \mathsf{B} = \mathcal{P}(Act \times \mathsf{Id}) \} \\
& \mathcal{P}(\mathsf{id} \times \backslash_k) \cdot \cup \cdot (\tau_l \times \tau_r) \cdot (\mathcal{P}(\mathsf{sub}_k \times \mathsf{id}) \cdot \omega \times \mathsf{id}) \times (\mathsf{id} \times \mathcal{P}(\mathsf{sub}_k \times \mathsf{id}) \cdot \omega) \cdot \Delta \\
= & \qquad \{ \text{ definition of } \alpha_k \} \\
& \mathcal{P}(\mathsf{id} \times \backslash_k) \cdot \cup \cdot (\tau_l \times \tau_r) \cdot ((\alpha_k \times \mathsf{id}) \times (\mathsf{id} \times \alpha_k)) \cdot \Delta
\end{aligned}$$

Consider, now, the right hand side of the same equation:

$$
\begin{aligned}
& \omega \cdot \boxplus \cdot (\backslash_k \times \backslash_k) \\
= \quad & \{ \boxplus \text{ is morphism } \} \\
& \mathcal{P}(\mathsf{id} \times \boxplus) \cdot \alpha_{\boxplus} \cdot (\backslash_k \times \backslash_k) \\
= \quad & \{ \text{ defintion of } \alpha_{\boxplus} \} \\
& \mathcal{P}(\mathsf{id} \times \boxplus) \cdot \cup \cdot (\tau_r \times \tau_l) \cdot ((\omega \times \mathsf{id}) \times (\mathsf{id} \times \omega)) \cdot \Delta \cdot (\backslash_k \times \backslash_k) \\
= \quad & \{ \Delta \text{ is natural, functors } \} \\
& \mathcal{P}(\mathsf{id} \times \boxplus) \cdot \cup \cdot (\tau_r \times \tau_l) \cdot ((\omega \cdot \backslash_k \times \backslash_k) \times (\backslash_k \times \omega \cdot \backslash_k)) \cdot \Delta \\
= \quad & \{ \backslash_k \text{ is morphism } \} \\
& \mathcal{P}(\mathsf{id} \times \boxplus) \cdot \cup \cdot (\tau_r \times \tau_l) \cdot ((\mathcal{P}(\mathsf{id} \times \backslash_k) \cdot \alpha_k \times \backslash_k) \times (\backslash_k \times \mathcal{P}(\mathsf{id} \times \backslash_k) \cdot \alpha_k)) \cdot \Delta \\
= \quad & \{ \text{ functors, } \tau_r \text{ and } \tau_l \text{ are natural } \} \\
& \mathcal{P}(\mathsf{id} \times \boxplus) \cdot \cup \cdot \mathcal{P}(\mathsf{id} \times (\backslash_k \times \backslash_k)) \times \mathcal{P}(\mathsf{id} \times (\backslash_k \times \backslash_k)) \cdot (\tau_r \times \tau_l) \\
& \cdot((\alpha_k \times \mathsf{id}) \times (\mathsf{id} \times \alpha_k)) \cdot \Delta \\
= \quad & \{ \cup \text{ is natural } \} \\
& \mathcal{P}(\mathsf{id} \times (\boxplus \cdot (\backslash_k \times \backslash_k))) \cdot \cup \cdot (\tau_r \times \tau_l) \cdot ((\alpha_k \times \mathsf{id}) \times (\mathsf{id} \times \alpha_k)) \cdot \Delta
\end{aligned}
$$

The simplification of both sides of equation (4.13) did not lead to the same expression. Actually, what we have concluded is that

$$\omega \cdot \boxplus \cdot (\backslash_k \times \backslash_k) = \mathcal{P}(\mathsf{id} \times (\boxplus \cdot (\backslash_k \times \backslash_k))) \cdot \gamma$$

and

$$\omega \cdot \backslash_k \cdot \boxplus = \mathcal{P}(\mathsf{id} \times (\backslash_k \cdot \boxplus)) \cdot \gamma$$

for coalgebra

$$\gamma = \cup \cdot (\tau_r \times \tau_l) \cdot ((\alpha_k \times \mathsf{id}) \times (\mathsf{id} \times \alpha_k)) \cdot \Delta$$

This means that both $\boxplus \cdot (\backslash_k \times \backslash_k)$ and $\backslash_k \cdot \boxplus$ are morphisms between γ and the final coalgebra ω. As there can only be one such morphisms we conclude they are equal.

□

This sort of proof is quite common in the calculus. The strategy is as follows: once a direct application of fusion is not possible, the aim becomes to show that both forms of composition of the two combinators can be defined as an anamorphism for a common gene coalgebra γ. Clearly, by the universal property, they must coincide. An important issue is the fact that γ was not postulated from the outset, but *inferred* from the calculations process.

4.3.2 The expansion law

A number of laws expressing equivalence between behaviours can be proved in this same calculational style. Reference [Bar01], for example, proves that both ⊞, ⊡ and ⊠ form Abelian monoids for whatever *interaction discipline* one might consider. Similarly, sum (*i.e.*, choice) is also an Abelian idempotent monoid. These results will be used in the sequel.

In this subsection we concentrate in proving the well-known expansion law. This law, a cornerstone in interleaving models of concurrency [Mil89], states a process is bisimilar to the sum of its immediate derivations, *i.e.*,

Lemma 4.1 *For all p,*

$$p \sim \sum_{p' \xrightarrow{a} p} a.p' \tag{4.14}$$

Proof. Recalling that a final coalgebra ω is always an isomorphism, the proof is as follows:

$$\begin{aligned}
& \omega\,(\sum_{p' \xrightarrow{a} p} a.p') \\
= & \qquad \{\text{ definition of } \xrightarrow{a}\} \\
& \omega\,(\sum_{\langle a,p'\rangle \in \omega\, p} a.p') \\
= & \qquad \{\text{ equation (4.10)}\} \\
& \bigcup \mathcal{P}\omega\,\{a.p' |\ \langle a, p'\rangle \in \omega\, p\} \\
= & \qquad \{\text{ definition of } \mathcal{P}\ \} \\
& \bigcup \{\omega\,(a.p') |\ \langle a, p'\rangle \in \omega\, p\} \\
= & \qquad \{\text{ equation (4.9)}\} \\
& \bigcup \{\{\langle a, p'\rangle\} |\ \langle a, p'\rangle \in \omega\, p\} \\
= & \qquad \{\ \cup \text{ reduction}\} \\
& \{\langle a, p'\rangle |\ \langle a, p'\rangle \in \omega\, p\} \\
= & \qquad \{\text{ functions}\} \\
& \omega\, p
\end{aligned}$$

□

4.3.3 Functional prototyping

One advantage of this approach to process algebra design is the fact that it allows an almost direct translation for a functional programming language like Haskell. This section highlights a few issues in the construction of a Haskell library for process algebra prototyping. Reference [BO02] reports on a first, alternative implementation on top of Charity [CF92, CS95].

Our starting point is the definition of the powerset functor `Pr` (assuming an implementation of sets as lists) and the definition of the semantic universe of processes as the coinductive type `Proc a`, as follows,

```
type Proc a = Nu (Pr a)
data Pr a x =  C [(a, x)]  deriving Show
instance Functor (Pr a)
    where fmap f (C s) = C (map (id >< f) s)

obsProc :: Pr a x -> [(a, x)]
obsProc p = f    where  (C f) = p

newtype Nu f = Fin (f (Nu f))
unFin :: Nu f -> f (Nu f)
unFin (Fin x) = x
```

The second step is the definition of the *interaction structure* as an *inductive* type, parametric on an arbitrary set of actions, over which one defines operator θ, denoted here as `prodAct`. To compare actions one must include in the class requirements a notion of action equality `eqAct`, expressed as the closure of an order relation `leAct`. For example, the Ccs interaction structure requires the following definition of *actions*:

```
data Act l = A l | AC l | Nop | Tau | Id  deriving Show
```

Dynamic combinators have a direct translation as functions over the final universe, as exemplified in the encoding of *prefix* and *choice*:

```
preP :: Act a -> Proc (Act a) -> Proc (Act a)
preP act p = Fin (C [(act,p)])

sumP ::  Proc (Act a) -> Proc (Act a) -> Proc (Act a)
sumP p q  = Fin (C (pp ++ qq)) where
            (C pp) = (unFin p)
            (C qq) = (unFin q)
```

On the other hand the definitons of static combinators are directly translated to HASKELL, provided that first one defines anamorphisms as a (generic) combinator. The following definition is standard:

```
ana :: Functor f => (c -> f c) -> c -> Nu f
ana phi = Fin . fmap (ana phi) . phi
```

Note, for example, how parallel composition | is defined in terms of the genes of ⊞ (alphai) and ⊠ (alphap):

```
par :: (Eq a) => (Proc (Act a), Proc (Act a)) -> Proc (Act a)
par (p, q) = ana alpha (p, q)
     where alpha (p, q) =
       C ((obsProc (alphai (p,q))) ++ (obsProc (alphap (p,q))))
```

4.4 Interruption and Recovery

4.4.1 Apomorphisms

This section introduces two *interruption* combinators, defined by *natural co-recursion*, and encoded as *apomorphisms* [VU97]. In this pattern the final result can be either generated in successive steps or 'all at once' without recursion. Therefore, the codomain of the source 'coalgebra' becomes the sum of its carrier with the coinductive type itself. The universal property is

The diagram is

$$\begin{tikzcd}
\nu_{\mathsf{T}} \arrow[r, "\omega_{\mathsf{T}}"] & \mathsf{T}\,\nu_{\mathsf{T}} \\
X \arrow[u, "\mathsf{apo}\,p"] \arrow[r, "p"'] & \mathsf{T}\,(X + \nu_{\mathsf{T}}) \arrow[u, "\mathsf{T}\,[\mathsf{apo}\,p{,}\mathsf{id}]"']
\end{tikzcd}$$

which entails the following universal property

$$h = \mathsf{apo}\,p \iff \omega_{\mathsf{T}} \cdot h = \mathsf{T}\,[h, \mathsf{id}] \cdot p \tag{4.15}$$

from which one can easily deduce the following *cancellation*, *reflection* and *fusion* laws.

$$\begin{aligned}
\omega_{\mathsf{T}} \cdot \mathsf{apo}\,\varphi &= \mathsf{T}[\mathsf{apo}\,\varphi, \mathsf{id}] \cdot \varphi && (4.16)\\
\mathsf{id} &= \mathsf{apo}\,\mathsf{T}(\iota_1) \cdot \omega_{\mathsf{T}} && (4.17)\\
\psi \cdot f = \mathsf{T}(f + \mathsf{id}) \cdot \varphi \Rightarrow \mathsf{apo}\,\psi \cdot f &= \mathsf{apo}\,\varphi && (4.18)
\end{aligned}$$

Note that every anamorphism $[\!(\varphi)\!]$ can be regarded as an apomorphism: $\mathsf{apo}\,\mathsf{T}(\iota_1) \cdot \varphi$. Dually, apomorphims reduce to anamorphisms composed with injections, *i.e.*, $\mathsf{apo}_{\mathsf{T}}\, p = [\!([p, \mathsf{T}\ \iota_2 \cdot \omega_{\mathsf{T}}])\!]_{\mathsf{T}} \cdot \iota_1$. Encoding apomorphisms in Haskell is straightforward:

```
apo :: Functor f => (c -> f (Either c (Nu f))) -> c -> Nu f
apo phi = Fin . fmap (either (apo phi) id) . phi
```

We shall, then, jump into the two announced applications.

4.4.2 Parallel Composition with Interruption

Our first combinator is a form of parallel composition which may terminate if some undesirable situation results from the interaction of the two processes. Such undesirable situation is abstractly represented by a particular form of interaction denoted by $*$. Therefore, combinator $\ddagger$ terminates execution as a result of an $*$-valued interaction. Formally, it is defined by an apomorphism

$$\ddagger \;=\; \mathsf{apo}\,\alpha_{\ddagger} \tag{4.19}$$

according to the following diagram

$$\begin{tikzcd}
\nu \times \nu \arrow[r, "\alpha_{\ddagger}"] \arrow[d, "\ddagger"'] & \mathcal{P}(Act \times ((\nu \times \nu) + \nu)) \arrow[d, "\mathcal{P}(\mathsf{id} \times [\ddagger, \mathsf{id}])"] \\
\nu \arrow[r, "\omega"] & \mathcal{P}(Act \times \nu)
\end{tikzcd}$$

where [1]

$$\begin{aligned}
\alpha_{\ddagger} \;=\; \nu \times \nu & \xrightarrow{\omega \times \omega} \mathcal{P}(Act \times \nu) \times \mathcal{P}(Act \times \nu) \\
& \xrightarrow{\mathcal{P}\tau_l \cdot \tau_r} \mathcal{P}\mathcal{P}((Act \times \nu) \times (Act \times \nu)) \\
& \xrightarrow{\mathcal{P}\mathsf{m} \cdot \cup} \mathcal{P}((Act \times Act) \times (\nu \times \nu)) \\
& \xrightarrow{\mathcal{P}(\theta \times \mathsf{id})} \mathcal{P}(Act \times (\nu \times \nu)) \\
& \xrightarrow{\mathcal{P}\mathsf{test}} \mathcal{P}(Act \times ((\nu \times \nu) + \nu))
\end{aligned}$$

[1] Note that, in this definition, strengths τ_l and τ_r are taken with respect to the powerset functor $\mathcal{P}$, in contrast with other combinators' definitions in this chapter where functor $\mathcal{P}(Act \times \mathsf{Id})$ is taken instead.

where

$$\mathsf{test} \;=\; \langle \pi_1, =_* \cdot \pi_1 \;\rightarrow\; \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \pi_2 \rangle$$

and $m : (A \times B) \times (C \times D) \longrightarrow (A \times C) \times (B \times D)$ is a natural isomorphism which exchanges the relative positions of factors in a product. Let us now illustrate how to compute with apomorphisms, by discussing the comutativity of this combinator, *i.e.*, that the following equation, where s is the comutativity isomorphism, holds.

$$\ddagger \cdot \mathsf{s} = \ddagger \tag{4.20}$$

As a first step we derive

$$\begin{array}{ll}
 & \ddagger \cdot \mathsf{s} = \ddagger \\
\equiv & \qquad \{\ \ddagger \text{ definition}\} \\
 & \mathsf{apo}\,\alpha_{\ddagger} \cdot \mathsf{s} \;=\; \mathsf{apo}\,\alpha_{\ddagger} \\
\Leftarrow & \qquad \{\ \text{apomorphism fusion law}\ \} \\
 & \alpha_{\ddagger} \cdot \mathsf{s} \;=\; \mathcal{P}(\mathsf{id} \times (\mathsf{s} + \mathsf{id})) \cdot \alpha_{\ddagger}
\end{array}$$

Now, let us unfold the left hand side of this last equality.

$$\begin{array}{ll}
 & \alpha_{\ddagger} \cdot \mathsf{s} \\
= & \qquad \{\ \alpha_{\ddagger} \text{ definition}\ \} \\
 & \mathcal{P}(\langle \pi_1, =_* \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \pi_2 \rangle) \cdot \mathcal{P}(\theta \times \mathsf{id}) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \cdot \tau_r \cdot \omega \times \omega \cdot \mathsf{s} \\
= & \qquad \{\ \mathsf{s} \text{ natural}\ \} \\
 & \mathcal{P}(\langle \pi_1, =_* \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \pi_2 \rangle) \cdot \mathcal{P}(\theta \times \mathsf{id}) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \cdot \tau_r \cdot \mathsf{s} \cdot \omega \times \omega \\
= & \qquad \{\ \tau_r \cdot \mathsf{s} = \mathcal{P}\mathsf{s} \cdot \tau_l\ \} \\
 & \mathcal{P}(\langle \pi_1, =_* \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \pi_2 \rangle) \cdot \mathcal{P}(\theta \times \mathsf{id}) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \cdot \mathcal{P}\mathsf{s} \cdot \tau_l \cdot \omega \times \omega \\
= & \qquad \{\ \tau_l \cdot \mathsf{s} = \mathcal{P}\mathsf{s} \cdot \tau_r, \text{ functors}\ \} \\
 & \mathcal{P}(\langle \pi_1, =_* \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \pi_2 \rangle) \cdot \mathcal{P}(\theta \times \mathsf{id}) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\mathcal{P}\mathsf{s} \cdot \mathcal{P}\tau_r \cdot \tau_l \cdot \omega \times \omega \\
= & \qquad \{\ \cup \text{ natural}\ \} \\
 & \mathcal{P}(\langle \pi_1, =_* \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \pi_2 \rangle) \cdot \mathcal{P}(\theta \times \mathsf{id}) \cdot \mathcal{P}\mathsf{m} \cdot \mathcal{P}\mathsf{s} \cdot \cup \cdot \mathcal{P}\tau_r \cdot \tau_l \cdot \omega \times \omega \\
= & \qquad \{\ \mathsf{m} \text{ natural: } \mathsf{m} \cdot \mathsf{s} = (\mathsf{s} \times \mathsf{s}) \cdot \mathsf{m}\ \} \\
 & \mathcal{P}(\langle \pi_1, =_* \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \pi_2 \rangle) \cdot \mathcal{P}(\theta \times \mathsf{id}) \cdot \mathcal{P}(\mathsf{s} \times \mathsf{s}) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_r \cdot \tau_l \cdot \omega \times \omega \\
= & \qquad \{\ \mathcal{P}\tau_r \cdot \tau_l = \mathcal{P}\tau_l \cdot \tau_r, \text{ because } \mathcal{P} \text{ is a commutative monad [Koc72]; functors}\ \} \\
 & \mathcal{P}(\langle \pi_1, =_* \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \pi_2 \rangle) \cdot \mathcal{P}((\theta \cdot \mathsf{s}) \times \mathsf{s}) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \cdot \tau_r \cdot \omega \times \omega \\
= & \qquad \{\ \times\text{-fusion}\ \} \\
 & \mathcal{P}(\langle \pi_1 \cdot ((\theta \cdot \mathsf{s}) \times \mathsf{s}), (=_* \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \pi_2) \cdot ((\theta \cdot \mathsf{s}) \times \mathsf{s}) \rangle) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \\
 & \cdot \tau_r \cdot \omega \times \omega \\
= & \qquad \{\ \text{conditional fusion, } \times\text{-cancellation, constant function}\ \} \\
 & \mathcal{P}(\langle \theta \cdot \mathsf{s} \cdot \pi_1, (=_* \cdot \theta \cdot \mathsf{s} \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \mathsf{s} \cdot \pi_2) \rangle) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \cdot \tau_r \cdot \omega \times \omega
\end{array}$$

Unfolding the right hand side we arrive at

$$
\begin{array}{ll}
 & \mathcal{P}(\mathsf{id} \times (\mathsf{s} + \mathsf{id})) \cdot \alpha_{\ddagger} \\
= & \qquad \{ \ \alpha_{\ddagger} \text{ definition } \} \\
 & \mathcal{P}(\mathsf{id} \times (\mathsf{s} + \mathsf{id})) \cdot \mathcal{P}(\langle \pi_1, =_* \cdot \pi_1 \to \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \pi_2 \rangle) \cdot \mathcal{P}(\theta \times \mathsf{id}) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \\
 & \cdot \tau_r \cdot \omega \times \omega \\
= & \qquad \{ \ \text{functors, } \times\text{-absorption } \} \\
 & \mathcal{P}(\langle \pi_1, (\mathsf{s} + \mathsf{id}) \cdot (=_* \cdot \pi_1 \to \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \pi_2) \rangle) \cdot \mathcal{P}(\theta \times \mathsf{id}) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \cdot \tau_r \cdot \omega \times \omega \\
= & \qquad \{ \ \text{conditional fusion } \} \\
 & \mathcal{P}(\langle \pi_1, =_* \cdot \pi_1 \to (\mathsf{s} + \mathsf{id}) \cdot \iota_2 \cdot \mathsf{nil}, (\mathsf{s} + \mathsf{id}) \cdot \iota_1 \cdot \pi_2 \rangle) \cdot \mathcal{P}(\theta \times \mathsf{id}) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \\
 & \cdot \tau_r \cdot \omega \times \omega \\
= & \qquad \{ \ +\text{-cancellation } \} \\
 & \mathcal{P}(\langle \pi_1, =_* \cdot \pi_1 \to \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \mathsf{s} \cdot \pi_2 \rangle) \cdot \mathcal{P}(\theta \times \mathsf{id}) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \cdot \tau_r \cdot \omega \times \omega \\
= & \qquad \{ \ \text{conditional fusion law, functors, } \times\text{-fusion } \} \\
 & \mathcal{P}(\langle \theta \cdot \pi_1 \, , \, =_* \cdot \theta \cdot \pi_1 \to \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \mathsf{s} \cdot \pi_2 \rangle) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \cdot \tau_r \cdot \omega \times \omega
\end{array}
$$

These two unfolding processes did not lead to the same expression; equation (4.20) is, therefore, in general false. Note, however, that the difference between the two expressions is only in the order in which the same arguments are supplied to θ. We may thus suppose the existence of a result weaker than (4.20), but still relevant and useful, may result from this calculation. This requires a more general discussion which follows.

4.4.3 Conditional Fusion

The aim of the previous calculation was to prove equation (4.20) which, by fusion, reduced to

$$\alpha_{\ddagger} \cdot \mathsf{s} \;=\; \mathcal{P}(\mathsf{id} \times (\mathsf{s} + \mathsf{id})) \cdot \alpha_{\ddagger} \tag{4.21}$$

Note the advantage of using a fusion law is to get rid of direct manipulation of recursion: all computation is done in terms of the recursion *genes*. In this way we succeeded in reducing (4.21) to

$$
\begin{array}{l}
\mathcal{P}(\langle \theta \cdot \mathsf{s} \cdot \pi_1 \, , \, (=_* \cdot \theta \cdot \mathsf{s} \cdot \pi_1 \to \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \mathsf{s} \cdot \pi_2) \rangle) \cdot \gamma \\
= \\
\mathcal{P}(\langle \theta \cdot \pi_1 \, , \, (=_* \cdot \theta \cdot \pi_1 \to \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \mathsf{s} \cdot \pi_2) \rangle) \cdot \gamma
\end{array}
$$

where

$$\gamma \;=\; \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \cdot \tau_r \cdot \omega \times \omega$$

Now note that this equation is only valid if one postulates an additional condition expressing the commutativity of θ, *i.e.*,

$$\theta \cdot \mathsf{s} \;=\; \theta \tag{4.22}$$

The interesting question is then: what does such a conditional validity at the *genes* level imply with respect to the validity of the original equation (4.20)? In general, suppose that in a proof, one concludes that the validity of the antecedent of the fusion law

$$\alpha \cdot f = \mathsf{T}(f + \mathsf{id}) \cdot \beta \;\;\Rightarrow\;\; \mathsf{apo}\,\alpha \cdot f \;=\; \mathsf{apo}\beta \tag{4.23}$$

depends on an additional condition Φ, *i.e.*,

$$\Phi \;\;\rightarrow\;\; \alpha \cdot f \;=\; \mathsf{T}(f + \mathsf{id}) \cdot \beta \tag{4.24}$$

What happens is that Φ is stated as a *local* condition on the genes of the apomorphisms, *i.e.*, on the imediate derivatives of the proocesses involved. Such a condition needs to be made stronger enforcing validity over *all* derivatives. Technically, Φ should be transformed into an *invariant*: *i.e.*, a predicate which is preserved by the coalgebra dynamics, ω, in the present case. To state such a result we need a modal language interpreted over coalgebras. The following notions are relatively standard in the literature (see, *e.g.*, [Mos99] or [Jac99]).

A predicate $\phi : U \longrightarrow \mathbf{2}$ over the carrier of a T-coalgebra $\langle U, \gamma : U \longrightarrow \mathsf{T}\ U\rangle$ is called a γ-*invariant* if closed under γ. Formally, one defines a predicate combinator $\bigcirc_\gamma$ [2]:

$$(\bigcirc_\gamma\ \phi)\ u \;\equiv\; \forall_{u' \in_{\mathsf{T}} \gamma\, u}\ .\ \phi\ u'$$

whose meaning reads: $\bigcirc_\gamma\ \phi$ is valid in all states whose immediate γ-derivatives verify ϕ. Then, ϕ is an invariant iif

$$\phi \;\rightarrow\; \bigcirc_\gamma\ \phi$$

that is

$$\phi \subseteq \bigcirc_\gamma\ \phi$$

The closure of $\bigcirc_\gamma$ defines the coalgebraic equivalent to the *always in the future* modal operator (just as $\bigcirc_\gamma$ corresponds to the *next* operator in modal logic). Thus, $\Box_\gamma\ \phi$ is introduced in [Jac99] as the *greatest* fixpoint of function $\lambda_x\,.\ \phi \cap \bigcirc_\gamma\ x$. Intuitively, $\Box_\gamma\ \phi$ reads 'ϕ holds in the current state and all the other states under γ. In such a definition contains the key to answer our previous question, as stated in the following lemma.

[2] Notation $\in_{\mathsf{T}}$ refers to the extension of the membership relation to regular functors [MB04].

Lemma 4.2 *Let α and β stand for two* T*-coalgebras and Φ a predicate over the carrier of β. Then,*

$$(\Phi \Rightarrow \alpha \cdot h = \mathsf{T}(h + \mathsf{id}) \cdot \beta) \;\; \rightarrow \;\; (\Box_\beta \, \Phi \rightarrow (\mathsf{apo}_\alpha \cdot h = \mathsf{apo}_\beta \mathsf{T})) \tag{4.25}$$

Proof. Let X be the carrier of β and i_Φ the inclusion in X of the subset classified by predicate Φ, *i.e.*, $\Phi \cdot i_\Phi = \underline{true} \cdot !$. Any β-invariant induces a subcoalgebra β' which makes $i_{\Box_\beta \, \Phi}$ a coalgebra morphism from β' to β. Then,

$$\begin{array}{cl}
& \Phi \Rightarrow \alpha \cdot h = \mathsf{T}(h + \mathsf{id}) \cdot \beta \\
\equiv & \qquad \{ \text{definition of inclusion } i_\Phi \} \\
& \alpha \cdot h \cdot i_\Phi = \mathsf{T}(h + \mathsf{id}) \cdot \beta \cdot i_\Phi \\
\Rightarrow & \qquad \{ \Box_\beta \Phi \subseteq \Phi \} \\
& \alpha \cdot h \cdot i_{\Box_\beta \Phi} = \mathsf{T}(h + \mathsf{id}) \cdot \beta \cdot i_{\Box_\beta \Phi} \\
\equiv & \qquad \{ i_{\Box_\beta \Phi} \text{ is a morphism from } \beta' \text{ to } \beta \} \\
& \alpha \cdot h \cdot i_{\Box_\beta \Phi} = \mathsf{T}(h + \mathsf{id}) \cdot \mathsf{T}(i_{\Box_\beta \Phi}) \cdot \beta' \\
\equiv & \qquad \{ \text{functors} \} \\
& \alpha \cdot h \cdot i_{\Box_\beta \Phi} = \mathsf{T}((h + \mathsf{id}) \cdot i_{\Box_\beta \Phi}) \cdot \beta' \\
\equiv & \qquad \{ \text{apomorfism fusion law} \} \\
& \mathsf{apo}\, \alpha \cdot h \cdot i_{\Box_\beta \Phi} = \mathsf{apo}\, \beta' \\
\equiv & \qquad \{ i_{\Box_\beta \Phi} \text{ is a coalgebra morphism} \} \\
& \mathsf{apo}\, \alpha \cdot h \cdot i_{\Box_\beta \Phi} = \mathsf{apo}\, \beta \cdot i_{\Box_\beta \Phi} \\
\equiv & \qquad \{ \text{inclusion } i_{\Box_\beta \, \Phi} \} \\
& \Box_\beta \Phi \Rightarrow (\mathsf{apo}\, \alpha \cdot h = \mathsf{apo}\, \beta)
\end{array}$$

□

We call formula (4.25) the *conditional fusion* law for apomorphisms. A similar result, but restricted to anamorphisms was proved in [Bar01]. Let's come back to our example. Note that in this case the relevant predicate, given by equation (4.22), does not involve states, but just *actions*. Therefore

$$\Box_\omega \, (\theta \cdot \mathsf{s} = \theta) \;=\; (\theta \cdot \mathsf{s} = \theta)$$

which, according to lemma 4.2, is the predicate to be used as the antecendent of (4.21). We may now conclude this example stating the following general law concerning the interruption operator:

$$(\theta \cdot \mathsf{s} = \theta) \;\; \rightarrow \;\; \ddagger \cdot \mathsf{s} = \ddagger \tag{4.26}$$

4.4.4 A Recovery Operator

We now discuss a combinator which models fault recovery[3]. Intuitively, the combinator allows the execution of its first argument until an *error* state is reached. By convention, an error occurrence is signalled by the execution of a special action x. When this is detected, execution control is passed to the second process. This process, which is the combinator second argument, is an abstraction for the system's recovery code. The combinator is defined as $\rhd \;=\; \mathsf{apo}\,\alpha_{\rhd}$, where

$$\begin{aligned}
\alpha_{\rhd} \;=\; \nu \times \nu & \xrightarrow{\omega \times \mathsf{id}} \mathcal{P}(Act \times \nu) \times \nu \\
& \xrightarrow{\mathsf{t}_x \cdot \pi_1 \rightarrow \iota_1, \iota_2 \cdot \pi_2} \mathcal{P}(Act \times \nu) \times \nu + \nu \\
& \xrightarrow{\tau_r + \omega} \mathcal{P}(Act \times (\nu \times \nu)) + \mathcal{P}(Act \times \nu) \\
& \xrightarrow{[\mathcal{P}(\mathsf{id} \times \iota_1), \mathcal{P}(\mathsf{id} \times \iota_2)]} \mathcal{P}(Act \times (\nu \times \nu + \nu))
\end{aligned}$$

where $\mathsf{t}_x : \mathcal{P}(Act \times \nu) \longrightarrow \mathbb{B}$ is given by

$$\mathsf{t}_x \;=\; \not\in_x \cdot \mathcal{P}\pi_1$$

We shall go on exploring the calculational power of this approach to process algebra through the discussion of a new conditional property. The intuition says that should no faults be detected in the first process, the recovery process will not be initiated. In other words, in the absence of faults, a process running in a fault tolerant environment behaves just as it would do if executed autonomously. Formally,

Lemma 4.3

$$\Box_{\omega}\,(\not\in_x \cdot \mathcal{P}\pi_1) \;\;\rightarrow\;\; \rhd \;=\; \pi_1 \tag{4.27}$$

Proof. Note that predicate $\not\in_x \cdot \mathcal{P}\pi_1$ only states fault absence in the immediate successors of each state. It is, therefore, sufficient to establish the antecedent of the fusion law, as shown below.

[3] Although the very abstract level in which it is approached here, it should be underlined that *fault tolerence* is a fundamental issue in software engineering.

$$\begin{array}{cl}
 & \mathcal{P}(\mathsf{id} \times (\pi_1 + \mathsf{id})) \cdot \alpha_{\triangleright} \\
= & \qquad \{ \text{ definition of } \mathsf{t}_x\text{, assuming hypothesis } \notin_x \bullet\mathcal{P}\pi_1 \ \} \\
 & \mathcal{P}(\mathsf{id} \times (\pi_1 + \mathsf{id})) \cdot [\mathcal{P}(\mathsf{id} \times \iota_1), \mathcal{P}(\mathsf{id} \times \iota_2)] \cdot (\tau_r + \omega) \cdot \iota_1 \cdot \omega \times \mathsf{id} \\
= & \qquad \{ \ \tau_r + \omega = [\iota_1 \cdot \tau_r, \iota_2 \cdot \omega]\} \\
 & \mathcal{P}(\mathsf{id} \times (\pi_1 + \mathsf{id})) \cdot [\mathcal{P}(\mathsf{id} \times \iota_1), \mathcal{P}(\mathsf{id} \times \iota_2)] \cdot [\iota_1 \cdot \tau_r, \iota_2 \cdot \omega] \cdot \iota_1 \cdot \omega \times \mathsf{id} \\
= & \qquad \{ \text{ +-cancellation } \} \\
 & \mathcal{P}(\mathsf{id} \times (\pi_1 + \mathsf{id})) \cdot \mathcal{P}(\mathsf{id} \times \iota_1) \cdot \tau_r \cdot \omega \times \mathsf{id} \\
= & \qquad \{ \ \mathcal{P} \text{ is a functor, +-cancellation, functors } \} \\
 & \mathcal{P}(\mathsf{id} \times \iota_1) \bullet \mathcal{P}(\mathsf{id} \times \pi_1) \bullet \tau_r \cdot \omega \times \mathsf{id} \\
= & \qquad \{ \ \mathcal{P}(\mathsf{id} \times \pi_1) \bullet \tau_r \ = \ \pi_1 \ \} \\
 & \mathcal{P}(\mathsf{id} \times \iota_1) \bullet \pi_1 \cdot \omega \times \mathsf{id} \\
= & \qquad \{ \ f \times g = \langle f, g \rangle, \times\text{-cancellation } \} \\
 & \mathcal{P}(\mathsf{id} \times \iota_1) \bullet \omega \cdot \pi_1
\end{array}$$

Note that what we would expect to have proven was

$$\mathcal{P}(\mathsf{id} \times (\pi_1 + \mathsf{id})) \cdot \alpha_{\triangleright} \ = \ \omega \cdot \pi_1$$

but, actually, all that was shown was that

$$\mathcal{P}(\mathsf{id} \times (\pi_1 + \mathsf{id})) \cdot \alpha_{\triangleright} \ = \ \mathcal{P}(\mathsf{id} \times \iota_1) \bullet \omega \cdot \pi_1$$

This comes to no surprise: the role of the additional factor $\mathcal{P}(\mathsf{id} \times \iota_1)$ in the right hand side is to ensure type compatibility between both sides of the equation. The important point, however, is the fact that the whole proof was carried under the assumption, recorded in the very first step, that $\notin_x \bullet\mathcal{P}\pi_1$. Thus, by lemma 4.2, we conclude as expected.

□

4.5 Behaviour-annotated interfaces

We end this chapter with a definition of what is to be called in the sequel a *behaviour-annotated interface*. As before, we assume a unique, general data domain, denoted by $\mathbb{D}$, as the type of all data values flowing in an application. Interfaces are defined over $\mathbb{D}$, but their behaviour is no long restricted to keeping track of port names and, possibly, of admissible types for data items flowing through them. Actually,

Definition 4.1 *Let $\mathbb{D}$ be a data domain understood as a general type for messages. An interface for a component C is specified by a* port signature*, $sig(C)$ over $\mathbb{D}$, given by a port name and a polarity annotation (either* $\mathsf{in}(put)$ *or* $\mathsf{out}(put)$*), and a* use pattern*, $use(C)$, given by a process term over port activations.*

In the context of component orchestration a *use pattern*, intended to abstract a component behaviour, is defined as follows:

Definition 4.2 *Let $\mathcal{P}$ be the set of port identifiers and S (the specification of) a component. Its use pattern, $use(S)$, is given by a process expression over $\mathbb{A}$ according to the grammar below, where $\mathbb{A}$ is defined as the union of the powerset of $\mathcal{P}$ with special symbol* 0 *(to represent inaction),*

$$P \ ::= \ \mathbf{0} \ \mid a \cdot P \mid P + P \mid P \boxtimes P \mid P \boxplus P \mid P \boxdot P \mid \sigma P \mid \mathsf{fix}\,(X \ = \ P)$$

*where a is an element of $\mathbb{A}$ (*i.e.*, a set of port identifiers) and σ is a substitution.*

Regarding as an action, a port identifier a asserts the activation of the corresponding port, i.e., the fact that a datum has crossed its boundaries. Note that choosing $\mathbb{A}$ as a *set* of port identifiers allows for the synchronous activation of several ports in a single computational step.

The semantics of such expressions is fairly standard, but for the parametrization of all forms of parallel composition (*i.e.*, $\boxtimes$ and $\boxdot$) by an interaction discipline as discussed above. Combinators $\mathbf{0}$, ., $+$, $\boxdot$, $\boxtimes$ and $\boxplus$, were already introduced. Renaming is given by term substitution. Expression $\mathsf{fix}\ (X \ = \ P)$ is a fixed point construction, which, as usual, can abbreviated in an explicit recursive definition.

As a basic principle underlying all models proposed in this thesis, direct interaction between components is precluded: all interaction are mediated by a specific connector. Therefore, if two components are active in a particular application, their joint behaviour will allow the realization of both use patterns either simultaneously or in an independent way. Formally,

Definition 4.3 *The joint behaviour of a collection $\{S_i | \ i \in n\}$ of components is given by*

$$use(S_1) \ \boxdot \ \ldots \ \boxdot \ use(S_n)$$

where the interaction discipline is fixed by $\theta = \cup$, i.e.*, the synchronisation of actions corresponds to the simultaneous realization of all of them. Clearly, the monoid unit is the emptyset $\emptyset$.*

This joint behaviour is computed by the application of the *expansion law* (4.14) proved above, while obeying to the interaction discipline given by θ. The following examples illustrate this construction.

Example 4.1 *Consider a component* S_1 *with two ports* a *and* b *whose use pattern is restricted to the activation of either* a *or* b*, forbidding their simultaneous occurrence. The expected behaviour is captured by*

$$use(S_1) \;=\; \mathsf{fix}\,(X \;=\; a \cdot X \;+\; b \cdot X)$$

Now consider another component, S_2*, with ports* c *and* d *whose behaviour is given by the co-occurrence of actions in both ports. Therefore,*

$$use(S_2) \;=\; \mathsf{fix}\,(X' \;=\; cd \cdot X'), \quad \textit{where} \;\; cd \;\overset{\text{abv}}{=}\; \{c, d\}$$

According to definition (4.3), the joint behaviour of S_1 *and* S_2 *is*

$$use(S_1) \;\boxdot\; use(S_2) \;=\; \mathsf{fix}\,(X \;=\; acd \cdot X \;+\; bcd \cdot X \;+\; a \cdot X \;+\; b \cdot X \;+\; cd \cdot X)$$

As a final example, consider still another component S_3*, with ports* e *and* f *activated in strict order,* i.e.*,*

$$use(S_3) \;=\; \mathsf{fix}\,(Y \;=\; e \cdot f \cdot Y)$$

Clearly, expansion leads to $use(S_2) \boxdot use(S_3) \;=\; P$*, where*

$$\begin{aligned} P &\;=\; \mathsf{fix}\,(X \;=\; cd \cdot X \;+\; e \cdot Q \;+\; cde \cdot Q) \\ Q &\;=\; \mathsf{fix}\,(X \;=\; cd \cdot X \;+\; f \cdot P \;+\; cdf \cdot P) \end{aligned}$$

In the next chapter we will introduce behaviour annotations also for software connectors and discuss the disciplines underlying their composition and interaction with components.

Chapter 5

Context-aware Connectors and Architectural Configurations

Summary

This chapter presents yet another model for software connectors with explicit behavioural annotations. The model was designed to deal correctly with context dependent behaviour and its propagation. Actually, capturing context dependent behaviour and ensuring it is suitably propagated through complex connector networks, is a difficult problem, still not addressed, for example, in the most popular semantics for REO. *A second contribution of the paper is a notion of* configuration, *which abstracts away fragments of a software architecture, putting together connectors and components, the latter specified by public interfaces in the style discussed in chapter 4. The chapter is based on [BB09], which formalises some of the constructions originally suggested in [BB06].*

5.1 Interfaces, connectors and configurations

Up to this chapter we have considered models for software connectors understood as special devices intended to regulate the flow of data, by relating data items crossing its input and output ports, and enforce synchronization constraints. Typically the coordinated entities, or *components*, are regarded as black-boxes, characterized by a set of ports through which data values are sent or received. Ports have a polarity (either *input* or *output*) and, maybe, a type to classify the admissible values. In general, our models made few assumptions on components.

As discussed in chapter 4, this is clearly insufficient to count as an *interface* for, for example, a web service. Typically, the latter includes a description of what is commonly called the *business protocol* or *behavioural pattern*, i.e., a specification of which, when and under what conditions ports become activated (*i.e.*, ready to deliver or consume a datum). To be useful such specifications have to be *compositional*, in the sense that the overall behaviour of an application should be computed from the behaviour of individual components and that of the *connectors* forming the coordination layer. The contribution of chapter 4 concerned precisely the definition of such enriched interfaces, building a formalism that can be used both for specifying component interfaces, as discussed there, and connector behaviours, as proposed in the present chapter.

Actually, the purpose of the present chapter is twofold:

- To define a framework for specification of coordination-based software architectures, putting together connectors and components, the latter exclusively accessed through behavioural interfaces. The key concept to be discussed is that of a *configuration*, as an abstraction of a piece of software architecture.
- To introduce yet another formal model for software connectors which, differently from the models discussed so far, allows the specification of connectors with *context dependent* behaviour.

Actually, a specific feature of the connector model to be proposed in the sequel is the ability to deal with *context-awareness*. By this we mean the ability of a connector to adapt its behaviour, in a non monotonic way, in response to context changes. Context is formed by the processes running in the coordinated components, as captured by the pending activity at the connector ports. This notion, even if in general is difficult to formalise, is regarded as fundamental in exogenous coordination research and as such is considered in the informal semantics of Reo as presented in its foundational papers [Arb03, Arb04].

Capturing context dependent behaviour and ensuring it is suitably *propagated* through a connector network, is a difficult problem. In the context of Reo it was not solved in the two main formal semantics proposed for the language based on timed data streams [AR03] and constraint automata [BSAR06], and remains, up to the time of writing a debatable issue. A partial solution was put forward in the so-called connector-coulouring semantics [CCA07], which is highly operational and admits degenerated behaviour in a number of cases. A semantics for Reo context-aware

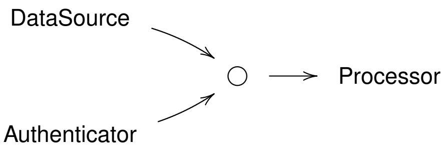


Figure 5.1: An example of a *configuration*.

connectors was announced as an outcome of a PhD thesis by David Costa [Cos10] but not yet published.

The development of a connector model dealing with context-awareness was put forward as main challenge back in 2004, at the proposal of this thesis. The outcome appeared in [BB09], as a follow up to ideas originally presented at FOCLASA'06 [BB06], and, in a revised version, in the present chapter.

A motivation example. Before jumping to the technical details, let us illustrate the envisaged strategy by means of a small example on web services coordination. Consider, thus, the situation depicted in Fig. 5.1 involving three web services: one produces data items, another emits authentication certificates for them and, finally, the third collects both data and certificates and processes them in a certain way.

According to the *World Wide Web Consortium*, a web service is a software application identified by a uniform resource identifier (URI), whose interfaces and binding can be defined, described, and discovered by XML artifacts, and that supports direct interactions with other software applications using XML based messages via Internet-based protocols. Less biased definitions abstract from concrete representations of data and messages. In [IBM03] a Web service is a *self-contained, modular applications that can be described, published, located, and invoked over a network, generally the Web.* And in [ACKM04] it is characterised as *a program accessible over the Web with a stable interface, published with additional descriptive information on some service directory.* At an even more abstract level, the underlying notion of a *service* emerges as a platform-independent computational entity which can be defined, published, classified, discovered and dynamically assembled for developing distributed, interoperable, evolvable systems and applications.

In general, services make themselves available by publishing an *interface* which describes a number of operations that other services may invoke according to given patterns, known as *conversations*, whose collection forms what is called the service *business protocol.* In the case of Web services, interfaces are typically described in WSDL [W2C07], which resemble a classical IDL ('interface description language') enriched with contextual information, such as the service address or the transport protocol used for access. The definition of admissible behaviours and the roles associated to them, usually resorts to a *coreography* language — e.g. Ws-Cdl [W2C05] It should also be noted that, unlike conventional middleware, a service can proactively initiate an interaction, a fact that blurs the classical distinction between clients and servers.

As in the previous chapters, we assume services are black box entities, accessed by purely syntactic interfaces. Moreover, such black boxes encapsulate active entities which are responsible for producing or consuming data items through their boundaries. The primary role of an interface is to keep track of port names and, possibly, of admissible types for data items flowing through them. The model, however, extends interfaces with a protocol specification over port activations, as formally defined in chapter 4.

For the moment, let us suppose each service in Fig. 5.1 has a particular protocol attached and moreover that there exists an overall restriction specifying that data from DataSource and Authenticator are received by the processing service in strict alternation (i.e., each data item is followed by the corresponding certificate).

The latter restriction clearly belongs to the coordination layer (the *glue code*): in principle the two data sources not even need to be aware of it. Enforcing such restrictions is the role of connectors which mediate services' interconnection. Connectors have ports through which the exchange of messages takes place. The activation of their ports also obey particular patterns, whose specification resort to the same formalism used for business protocols. When interfacing with services both sides impose constraints on how the conversation has to proceed. The purpose of the model discussed in the sequel is to be able to infer the global behaviour of an application from such constraints.

As detailed in section 5.2, connectors are specified by a relation, which captures the flow of data, and a behavioural pattern. Their construction is compositional: new connectors are built out of old through six specific connectives: parallel and concurrent *aggregation*, *interleaving*, left and right port *join* and *hook*, the latter corresponding, as before, to a feedback mechanism.

Connectors provide the essential mechanism for service composition. This guarantees loosely coupled cooperation among services and entails a number of simplifications. For example there is no need to syntactically distinguish between different forms of interaction, *e.g.*, between *one-way* or *request-response* interactions. Such patterns are enforced at the coordination level. A collection of services, specified by their interfaces, interconnected by a particular, eventually rather complex connector, forms a *configuration*, as depicted in Fig. 5.1.

The remaining sections are devoted to make all this precise. The *glue code* is discussed in sections 5.2, 5.3, and 5.4, which are concerned with connectors and their composition into a coordination layer. Finally, section 5.5 explains how connectors and services (or components) are put together to cooperate in a loosely-coupled way.

5.2 The coordination layer

The fundamental notion proposed in this model as a basis for component orchestration is that of a *configuration*. As explained above, this captures the intuition that components cooperate through specific *connectors* which abstracts the idea of an intermediate *glue code* to handle interaction. Having defined, in the previous chapter, a notion of component *interface*, which records all what may be assumed to be known by potential clients, we shall now complete the picture by defining

- what *connectors* are and how do they compose;
- how do services' *interfaces* and *connectors* interact in a configuration.

These points are tackled in the following sections. It turns out that two forms of composition (of connectors with themselves and with components' interfaces) follow *different* interaction disciplines, captured by specific definitions of θ.

5.2.1 Connectors

Connectors are *glueing devices* between services which ensure the flow of data and the meet of synchronization constraints. Their specification builds on top of the models presented earlier in the thesis, which are extended here with an explicit annotation of activation, or *use*, patterns for their *ports*.

Ports are *interface points* through which messages flow. Each port has an *interaction polarity*, either *input* or *output* (a *source* or a *sink* end in the Reo terminology), but, in general, connectors are blind with respect to the data values flowing through them.

Let, as usual, $\mathbb{C}$ be a connector with m input and n output ports. Assume $\mathbb{M}$ is a generic type of messages and $\mathbb{P}$ a set of (unique) *port identifiers*. In a number of cases it is necessary to consider a default value in $\mathbb{D}$ to represent *absence* of messages, for example to describe a transition in a connector's state in which a particular port is not involved. Therefore, the type of data $\mathbb{D}$ flowing through connectors is defined as

$$\mathbb{D} \triangleq \mathbb{M} + \mathbf{1} \tag{5.1}$$

where $\mathbf{1}$ is the singleton set whose unique element is represented, by convention, as $\perp$.

Elementary connectors are stateless, but to introduce asynchrony, *e.g.*, through a buffered channel, internal states might be considered. Let $\mathbb{U}$ stand for a generic type of state spaces, typically given as a functorial expression in $\mathbb{D}$. For example $\mathbb{U}$ may be defined as a sequence of data ($\mathbb{U} = \mathbb{D}^*$) or, as in the specification of a one-fifo buffer below, simply as $\mathbb{U} = \mathbb{D}$. Default value $\perp$ standing now for absence of stores information in the connectors memory. Formally, the behaviour of a connector is defined as follows

Definition 5.1 *The specification of a connector* $\mathbb{C}$ *is given by a relation*

$$\mathsf{data}.[\![\mathbb{C}]\!] : \mathbb{D}^n \times \mathbb{U} \longleftarrow \mathbb{D}^m \times \mathbb{U} \tag{5.2}$$

which records the flow of data, and a process expression

$$\mathsf{port}.[\![\mathbb{C}]\!] \in Bhv \tag{5.3}$$

which gives the behavioural pattern for port activation. Bhv is the process language generated according to the grammar in definition 4.2.

In a first approximation set $\mathbb{A}$, of actions, above is taken as $\mathbb{A} = \mathcal{P}(\mathbb{P} \cup \{\tau\})$, *i.e.*, as sets of connectors' ends plus a special symbol, τ, to represent any unobservable action. The introduction of τ is technically entailed by the semantics of the *hook* combinator, as explained below. Regarded as an action, a port identifier a asserts the activation of the corresponding port, i.e., the fact that a datum crosses its boundaries. Note that choosing A as a *set* of port identifiers allows for the synchronous activation of several ports in a single computational step. This is enough for the semantics of a number of elementary connectors, as follows.

Synchronous channel.

The *synchronous channel* has two ports of opposite polarity. This connector forces input and output to become mutually blocking, in the sense that any of them must wait for the other to be completed.

$$\begin{aligned}
\mathsf{data}.[\![\ a \longrightarrow b\ ]\!] &= \mathsf{Id}_{\mathbb{D}\times\mathbb{U}} \\
\mathsf{port}.[\![\ a \longrightarrow b\ ]\!] &= \mathsf{fix}\ (X = ab \cdot X)
\end{aligned}$$

Here, as well as in the next three cases, state information is irrelevant. Therefore, $\mathbb{U} = \mathbf{1}$. Its semantics is simply the identity relation on data domain $\mathbb{D}$ and its behaviour is captured by the simultaneous activation of its two ports.

Unreliable channel.

Any coreflexive relation, that is any subset of the identity, provides channels which can loose information, thus modelling unreliable communications. Therefore, we define, an *unreliable channel* as

$$\begin{aligned} \mathsf{data}.\llbracket\, a \xrightarrow{\cdots} b \,\rrbracket &\subseteq \mathsf{Id}_{\mathbb{D}\times\mathbb{U}} \\ \mathsf{port}.\llbracket\, a \xrightarrow{\cdots} b \,\rrbracket &= \mathsf{fix}\ (X \ = ab \cdot X + a \cdot X) \end{aligned}$$

The behaviour is given by a choice between a successful communication, represented by the simultaneous activation of the ports or, by a failure, represented by the single activation of the input port.

Filter channel.

This is a channel in which some messages are discarded in a controlled way, according to a given predicate $\phi : 2 \longleftarrow \mathbb{D}$. Therefore, all messages failing to verify ϕ are lost. Regarding predicate ϕ as a relation $\mathsf{R}_\phi : \mathbb{D} \times \mathbb{U} \longleftarrow \mathbb{D} \times \mathbb{U}$ such that

$$(d, \bot)\mathsf{R}_\phi(d', \bot) \text{ iff } d = d' \ \wedge \ (\phi\, d)$$

define

$$\begin{aligned} \mathsf{data}.\llbracket\, a \xrightarrow{\phi} b \,\rrbracket &= \mathsf{R}_\phi \\ \mathsf{port}.\llbracket\, a \xrightarrow{\phi} b \,\rrbracket &= \mathsf{fix}\ (X = ab \cdot X + a \cdot X) \end{aligned}$$

Drain.

A drain has two input, but no output, ports. Therefore, it looses any data item crossing its boundaries. A drain is *synchronous* if both write operations are requested to succeed at the same time (which implies that each write attempt remains pending until another write occurs in the other extremity). It is *asynchronous* if, on the other hand, write operations in the two ports do not coincide. The formal definitions are, respectively,

$$
\begin{aligned}
\mathsf{data}.[\![\; a \vdash\!\!\!\longrightarrow\!\!\!\dashv b \;]\!] &= (\mathbb{D} \times \mathbb{U}) \times (\mathbb{D} \times \mathbb{U})\\
\mathsf{port}.[\![\; a \vdash\!\!\!\longrightarrow\!\!\!\dashv b \;]\!] &= \mathsf{fix}\ (X = ab \cdot X)
\end{aligned}
$$

and

$$
\begin{aligned}
\mathsf{data}.[\![\; a \overset{\triangledown}{\vdash\!\!\!\longrightarrow\!\!\!\dashv} b \;]\!] &= (\mathbb{D} \times \mathbb{U}) \times (\mathbb{D} \times \mathbb{U})\\
\mathsf{port}.[\![\; a \overset{\triangledown}{\vdash\!\!\!\longrightarrow\!\!\!\dashv} b \;]\!] &= \mathsf{fix}\ (X = a \cdot X + b \cdot X)
\end{aligned}
$$

Fifo$_1$.

This is a channel with a buffer of a single position. Thus $\mathbb{U} = \mathbb{D}$

$$
\begin{aligned}
\mathsf{data}.[\![\; a \vdash\!\!-\Box\!\!\rightarrowtail b \;]\!] &= \mathsf{R}_{\Box}\\
\mathsf{port}.[\![\; a \vdash\!\!-\Box\!\!\rightarrowtail b \;]\!] &= \mathsf{fix}\ (X = a \cdot b \cdot X)
\end{aligned}
$$

where $\mathsf{R}_{\Box}$ is given by the following clauses, for all $d, u \in \mathbb{D}$,

$$(\perp, d)\ \mathsf{R}_{\Box}\ (d, \perp) \tag{5.4}$$

$$(u, \perp)\ \mathsf{R}_{\Box}\ (\perp, u) \tag{5.5}$$

Clause (5.4) corresponds to the effect of an input at port a, whereas clause (5.5) captures output at port b, which requires the presence of a datum in the internal state. Notice that clause (5.4) precludes input whenever the buffer is full. An *eager* alternative overwrites the buffer's memory:

$$(\perp, d)\ \mathsf{R}_{\Box}\ (d, u) \tag{5.6}$$

Similarly, clause (5.5) defines b as what is often called a *get* port: data is read and removed from the connector. Alternatively, a single *read* port can be specified by

$$(u, u)\ \mathsf{R}_{\Box}\ (\perp, u) \tag{5.7}$$

In a number of practical situations component orchestration depends not only on port activation, but also on the absence of service requests at particular ports in configuration. A typical example is provided by one of the basic channels in Reo: the *lossy channel*, which acts as a synchronous one if both an input and an output requests are pending on its source and sink ends, respectively, but looses any data item on input on the absence of an output request in the other end. Notice this behaviour is distinct from that of the *unreliable* channel, which looses data non deterministically.

To handle these cases we enrich the specification of the set of actions A to include *negative port activations* , or more rigorously stated, absence of port requests, denoted, for each port p, by $\tilde{p}$. Technically, actions are given by datatype

$$\mathbb{A} = \mathcal{P}(\mathbb{P} \cup \{\tau\}) \times \mathcal{P}\mathbb{P} \tag{5.8}$$

subject to the following invariant

$$\mathsf{disjoint}\langle pos, neg\rangle \;=\; (pos \cap neg = \emptyset) \tag{5.9}$$

Moreover, *absence of port information* is only relevant to *output* ports, which may carry, or not, a request for receiving information. Therefore, if a is an *input* port,

$$a \;= \tilde{a} \tag{5.10}$$

Values of type $\mathbb{A}$ are represented according to the following abbreviation

$$\langle\{a,b,c\},\{d,f\}\rangle \overset{\text{abv}}{=} abc\widetilde{df} \tag{5.11}$$

Therefore, the specification of a lossy channel becomes

Lossy.

$$\mathsf{data}.[\![\, \bullet \dashrightarrow \bullet \,]\!] \;\subseteq\; \mathsf{Id}_{\mathbb{D}} \tag{5.12}$$

$$\mathsf{port}.[\![\, \bullet \dashrightarrow \bullet \,]\!] \;=\; \mathsf{fix}(X \;=\; ab \cdot X + a\tilde{b} \cdot X) \tag{5.13}$$

where $\tilde{b}$ corresponds to an absence of a reading request at port b. A lossy channel only transmits if a potential receiver is asking for the data item sent. Otherwise, data is lost.

5.2.2 New connectors from old

Complex connectors are built out of simpler ones through a set of six *combinators*: *parallel* and *concurrent* composition, *interleaving*, *hook*, *left join* and *right join*. The explicit use of combinators provides a structural alternative to composition through graph manipulation, used, for example, in mainstream Reo literature.

In the sequel, let $t_{\#a}$, for $t \in \mathbb{D}^n \times \mathbb{U}$ and $a \in \mathcal{P}$, denote the component of data tuple t corresponding to port a, and $t_{|a}$ a tuple identical to t from which component $t_{\#a}$ has been deleted. Note that formal definitions of these operators were introduced in chapter 2.

Aggregation.

There are three combinators, denoted by $\boxplus$, $\boxtimes$ and $\boxdot$, whose effect is to place their arguments side-by-side, with no direct interaction between them. They distinguish one of the other by the way the arguments's behavioural patterns are combined: through *parallel* composition or *interleaving*, respectively. Formally,

$$\mathsf{port}.\llbracket \mathbb{C}_1 \boxplus \mathbb{C}_2 \rrbracket = \mathsf{port}.\llbracket \mathbb{C}_1 \rrbracket \boxplus \mathsf{port}.\llbracket \mathbb{C}_2 \rrbracket \tag{5.14}$$

$$\mathsf{port}.\llbracket \mathbb{C}_1 \boxtimes \mathbb{C}_2 \rrbracket = \mathsf{port}.\llbracket \mathbb{C}_1 \rrbracket \boxtimes \mathsf{port}.\llbracket \mathbb{C}_2 \rrbracket \tag{5.15}$$

$$\mathsf{port}.\llbracket \mathbb{C}_1 \boxdot \mathbb{C}_2 \rrbracket = \mathsf{port}.\llbracket \mathbb{C}_1 \rrbracket \boxdot \mathsf{port}.\llbracket \mathbb{C}_2 \rrbracket \tag{5.16}$$

taking, in the first two cases, $\theta = \cup$ to capture the envisaged interaction discipline.

At data level all combinators behave as a relational product upon some re-arranging to separate state from data information. Such housekeeping task is done by Set-isomorphism $\mathsf{m} \;:\; (A \times B) \times (C \times D) \;\cong\; (A \times C) \times (B \times D)$. Formally,

$$\begin{array}{c} \mathsf{data}.\llbracket \mathbb{C}_1 \boxplus \mathbb{C}_2 \rrbracket = \mathsf{data}.\llbracket \mathbb{C}_1 \boxtimes \mathbb{C}_2 \rrbracket = \mathsf{data}.\llbracket \mathbb{C}_1 \boxdot \mathbb{C}_2 \rrbracket \\ = \\ \mathsf{data}.\llbracket \mathbb{C}_1 \rrbracket \Box\, \mathsf{data}.\llbracket \mathbb{C}_2 \rrbracket \end{array}$$

with

$$R \Box S \;=\; \mathsf{m} \cdot (R \times S) \cdot \mathsf{m} \tag{5.17}$$

which, going pointwise, amounts to

$$((\vec{d'}, \vec{e'}), (u', v'))\; R \Box S \;((\vec{d}, \vec{e}), (u, v)) \;\equiv\; (\vec{d'}, u')\; R\; (\vec{d}, u) \;\wedge\; (\vec{e'}, v')\; S\; (\vec{e}, v)$$

Combinators $\boxtimes$ and $\boxdot$, to be denoted by $\otimes$ and $\odot$, respectively, which enforce interaction between their arguments, admit a *strong* version inn order to be able to propagate negative information.

Let us consider first $\otimes$, the strong synchronous product. The intuition underlying the definition of $P \otimes Q$is as follows: whenever a term in the expansion of P has a negative port $\tilde{p}$ (and recall that by (5.10) p has an *output* polarity), it must be *multiplied*, through $\boxtimes$, by $\tilde{P}$ to prepare the grounds for propagating the associated negative information. This may, in particular, turn *negative* an output port in P which may represent the propagation of negative information. Examples will be given soon after the introduction of the *hook* combinator. Formally, let $\Upsilon(P)$ denote the immediate expansion of behavioural expression P, *i.e.*,

$$\Upsilon(P) \;=\; \sum_{i \in \{1, \cdots, m\}} \omega_i \cdot P_i \tag{5.18}$$

such that $P \sim \Upsilon(P)$. Each summand F in $\Upsilon(P)$ is said to be a factor of P, a fact we represent by $F \leftarrow \Upsilon(P)$. Let $F = \omega \cdot R$ such that

$$\omega \cdot R \leftarrow \Upsilon(P) \;\text{ and }\; \omega \cap \widetilde{\mathbb{A}} \neq \emptyset$$

In this case F is said to be a factor of P with negated ports and represented by $F \leftarrow \Upsilon_n(P)$. Now define $\otimes$ as

$$\otimes \;=\; [\!(\alpha_\otimes)\!] \tag{5.19}$$

with

$$\alpha_\otimes(P,Q) \;= \alpha_\boxtimes(P,Q) \cup \bigcup_{\substack{\omega \cdot P' \leftarrow \Upsilon_n(P) \\ j \in \{1,\cdots,n\}}} (\omega \cup \widetilde{\omega_j}, (P', Q_j)) \cup \bigcup_{\substack{\omega \cdot Q' \leftarrow \Upsilon_n(Q) \\ i \in \{1,\cdots,m\}}} (\widetilde{\omega_i} \cup \omega, (P_i, Q'))$$

for $\Upsilon(P) = \sum_{i \in \{1,\cdots,m\}} \omega_i \cdot P_i$ and $\Upsilon(Q) = \sum_{j \in \{1,\cdots,n\}} \omega_j \cdot Q_j$.

Note that (5.19) is the (coinductive) gene of the following explicitly recursive definition:

$$P \otimes Q \;=\; P \boxtimes Q \;+ \sum_{\substack{\omega \cdot P' \leftarrow \Upsilon_n(P) \\ j \in \{1,\cdots,n\}}} (\omega \cup \widetilde{\omega_j}) \cdot (P' \otimes Q_j) \;+ \sum_{\substack{\omega \cdot Q' \leftarrow \Upsilon_n(Q) \\ i \in \{1,\cdots,m\}}} (\widetilde{\omega_i} \cup \omega) \cdot (P_i \otimes Q') \tag{5.20}$$

Clearly, if neither P nor Q have negative factors $P \otimes Q = P \boxtimes Q$.

The strong version of parallel composition is defined by combining, at the *genes* level, the effects of both $\boxplus$ and $\otimes$, just as $\boxdot$ in (4.11) combines $\boxplus$ and $\boxtimes$. Formally,

$$\odot \;=\; [\!(\alpha_\odot)\!]$$

where

$$\begin{aligned} \alpha_\boxdot \;=\; & \nu \times \nu \xrightarrow{\;\triangle\;} (\nu \times \nu) \times (\nu \times \nu) \xrightarrow{\;(\alpha_\boxplus \times \alpha_\otimes)\;} \\ & \mathcal{P}(\mathbb{A} \times (\nu \times \nu)) \times \mathcal{P}(\mathbb{A} \times (\nu \times \nu)) \xrightarrow{\;\cup\;} \mathcal{P}(\mathbb{A} \times (\nu \times \nu)) \end{aligned}$$

Note that ν is just, as in chapter 4, the carrier of the final coalgebra for this functor.

Hook.

This combinator encodes a *feedback* mechanism, drawing a direct connection between an output and an input port. This has a double consequence: the connected ports must be activated simultaneously and become externally non observable. Formally, such conditions must be expressed in $\mathsf{port}.[\![\mathbb{C} \ \Lsh^{j}_{i}]\!]$ and their specification requires some care.

The crucial issue is the suitable definition of a new combinator for behaviours, $\mathsf{hide}\, c$, parametric on a set $c \subseteq \mathbb{A} - \{0\}$, whose effect is to prune its argument according to the following rules

- all computations exhibiting occurrences of non empty strict subsets of c must be removed, because ports in c have be activated simultaneously;
- there is, however, an exception to the rule above: if a computation exhibits a non empty strict subsets of c, c' such that c' only contains negative *output* ports, a property denoted by $\mathsf{negfac}(c')$, then such a computation is not removed.

The intuition for the last rule is that if in the only occurrence of an *output* port which the hook combinator aims to internalise, is negative, *i.e.*, has, in the computation considered, no pending output request, it can be ignored: there is no matching input port, but also no information to be transmitted.

The combinator then hides all references to c in the remaining computations, either by removing them when occurring in a strictly larger context or by mapping them to an unobservable action τ when occurring isolated. It is defined as

$$\mathsf{hide}\, c \;=\; [\!(\alpha_{\mathsf{hide}\, c})\!] \tag{5.21}$$

where

$$\alpha_{\mathsf{hide}\, c} \;=\; \nu \xrightarrow{\omega} \mathcal{P}(\mathbb{A} \times \nu) \xrightarrow{\mathsf{h}_c} \mathcal{P}(\mathbb{A} \times \nu)$$

and

$$\begin{aligned} \mathsf{h}_c\ s \;&= \{\langle a \setminus c, u\rangle |\ \langle a, u\rangle \in s \ \wedge\ ((a \cap c \neq \emptyset) \ \rightarrow\ (c \subset a \ \vee\ \mathsf{negfac}(a \cap c)))\} \\ &\cup\ \{\langle \tau, u\rangle |\ \langle c, u\rangle \in s\} \end{aligned}$$

Thus, let i, respectively j, be an output, respectively, input, port in connector $\mathbb{C}$. The *hook* combinator links i to j according to the following definition:

$$\mathsf{port}.[\![\mathbb{C} \ \Lsh^{j}_{i}]\!] \;=\; \mathsf{hide}\,\{i, j\}\,\mathsf{hide}\,\{\widetilde{i}, \widetilde{j}\}\,\mathsf{port}.[\![\mathbb{C}]\!]$$

If $\mathsf{data}.[\![\mathbb{C}]\!] : \mathbb{D}^n \times U \longleftarrow \mathbb{D}^m \times U$, the effect of *hook* on the data flow relation is modelled by relation

$$\mathsf{data}.[\![\mathbb{C} \Lsh^j_i]\!] : \mathbb{D}^{n-1} \times U \longleftarrow \mathbb{D}^{m-1} \times U$$

$$t'_{|j}\,(\mathsf{data}.[\![\mathbb{C} \Lsh^j_i]\!])\,t_{|i} \quad \text{iff} \quad t'\,(\mathsf{data}.[\![\mathbb{C}]\!])\,t \;\wedge\; t'_{\#j} = t_{\#i}$$

Example 5.1 *Let us illustrate the hook combinator through two elementary examples, which do not involve negative information. More complex examples are discussed in next section, in which it is shown the suitability of this combinator to handle composition of context-aware connectors. For the moment, consider connectors $\mathbb{C}$ and $\mathbb{F}$, both with an input and an output port, named a, a' in the first case, and b, b' in the second. Let us analyse composition $(\mathbb{C} \odot \mathbb{F}) \Lsh^b_{a'}$. At the data level, one gets*

$$\begin{aligned}
& (y,(u',v'))\ \mathsf{data}.[\![(\mathbb{C} \odot \mathbb{F}) \Lsh^b_{a'}]\!]\ (x,(u,v)) \\
= \quad & \qquad \{\ \textit{unfolding definitions}\ \} \\
& \exists_z\ .\ \ ((z,y),(u',v'))\ \mathsf{data}.[\![\mathbb{C} \odot \mathbb{F}]\!]\ (x,z),(u,v)) \\
= \quad & \qquad \{\ \textit{unfolding definitions}\ \} \\
& \exists_z\ .\ \ (z,u')\ \mathsf{data}.[\![\mathbb{C}]\!]\ (x,u)\ \wedge\ (y,v')\ \mathsf{data}.[\![\mathbb{F}]\!]\ (z,v)
\end{aligned}$$

which shows that the hook combinator *encodes a form of relational composition which is* partial *in the sense that only part of the output is fed back as new input. For the behavioural component, consider $\mathbb{C}$ and $\mathbb{F}$ as synchronous channels. Then,*

$$\mathsf{port}.[\![(\mathbb{C} \odot \mathbb{F}) \Lsh^b_{a'}]\!] \;=\; \mathsf{fix}\ (x = ab'.x)$$

because the other two terms in the expansion $\mathsf{fix}\ (x = aa'.x + bb'.x + aa'bb'.x)$ *contain strict subsets of $c = \{a', b\}$. Note that the synchronous channel always acts as the identity for* hook. *Suppose, now, that $\mathbb{F}$ is defined as a $Fifo_1$ channel. Thus, and adopting, in the sequel, the convention which abbreviates* $\mathsf{port}.[\![\mathbb{C}]\!]$ *to $\mathbb{C}$,*

$$\begin{aligned}
& \mathsf{port}.[\![(\mathbb{C} \odot \mathbb{F}) \Lsh^b_{a'}]\!] \\
\sim \quad & \qquad \{\ \textit{hook definition and expansion law}\ \} \\
& aa'b \bullet (\mathbb{C} \odot b' \bullet \mathbb{F}) \Lsh^b_{a'} + aa' \bullet (\mathbb{C} \odot \mathbb{F}) \Lsh^b_{a'} + b \bullet (\mathbb{C} \odot \mathbb{F}) \Lsh^b_{a'} \\
\sim \quad & \qquad \{\ \mathsf{hide}\ \ \textit{definition}\ \} \\
& a \bullet (\mathbb{C} \odot b' \bullet \mathbb{F}) \Lsh^b_{a'} \\
\sim \quad & \qquad \{\ \textit{expansion law}\ \} \\
& a \bullet (aa' \bullet (\mathbb{C} \odot b' \bullet \mathbb{F}) \Lsh^b_{a'} + b' \bullet (\mathbb{C} \odot \mathbb{F}) \Lsh^b_{a'} + aa'b' \bullet (\mathbb{C} \odot \mathbb{F}) \Lsh^b_{a'})
\end{aligned}$$

$$\begin{array}{ll} \sim & \{\ \textsf{hide}\ \ \textit{definition}\ \} \\ & a \bullet b' \bullet (\mathbb{C} \odot \mathbb{F}) \Lsh^{b}_{a'} \\ \sim & \{\ \textit{introducing}\ \textsf{fix}\ \} \\ & \textsf{fix}\ (x = a \bullet b' \bullet x) \end{array}$$

Join.

The last combinator considered here is called *join* and its effect is to plug ports with identical polarity. The aggregation of *output* ports is done by a *right join* ($\mathbb{C}\ {}^{i}_{j} > z$), where $\mathbb{C}$ is a connector, i and j are ports and z is a fresh name used to identify the new port. Port z receives asynchronously messages sent by either i or j. When messages are sent at same time the combinator chooses one of them non deterministically.

On the other hand, aggregation of *input* ports resorts to a *left join* ($z <^{i}_{j} \mathbb{C}$). This behaves like a *broadcaster* sending synchronously messages from z to both i and j. Formally, for $\textsf{data}.[\![\mathbb{C}]\!] : \mathbb{D}^n \times U \longleftarrow \mathbb{D}^m \times U$, we define

Right join:
The data flow relation $\textsf{data}.[\![\mathbb{C}\ {}^{i}_{j} > z]\!] : \mathbb{D}^{n-1} \times U \longleftarrow \mathbb{D}^m \times U$ for this operator is given by

$$r\,(\textsf{data}.[\![\mathbb{C}\ {}^{i}_{j} > z]\!])\,t \quad \text{iff} \quad t'\,(\textsf{data}.[\![\mathbb{C}]\!])\,t \ \wedge\ r_{|z} = t'_{|i,j} \ \wedge\ (r_{\#z} = t'_{\#i} \ \vee\ r_{\#z} = t'_{\#j})$$

At the behavioural level, its effect is that of a renaming operation

$$\textsf{port}.[\![(\mathbb{C}\ {}^{i}_{j} > z)]\!] \ = \ \{z \leftarrow i, z \leftarrow j\}\ \textsf{port}.[\![\mathbb{C}]\!]$$

Example 5.2 *The merger connector depicted in Figure 5.2 is obtained by a right join of the sink end of two interleaved synchronous channels.*

$$\mathbb{M} \triangleq (\ a \longrightarrow a' \ \boxplus\ b \longrightarrow b'\ )\,{}^{a'}_{b'} > w \ =$$

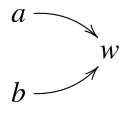


Figure 5.2: A *merger*.

Its behavioural pattern is

$$\textsf{port}.[\![\mathbb{M}]\!] \ = \ \textsf{fix}\ (x = aw.x + bw.x)$$

because,

$$
\begin{array}{ll}
 & \mathsf{port}.[\![\mathbb{M}]\!] \\
\sim & \quad \{ \textit{ definition of right join and synchronous channel } \} \\
 & \{w \leftarrow a', w \leftarrow b'\}\ (\mathsf{fix}\ (x\ =\ aa' \cdot x)\ \boxplus\ \mathsf{fix}\ (x\ =\ bb' \cdot x)) \\
\sim & \quad \{ \textit{ definition of interleaving and expansion law } \} \\
 & \{w \leftarrow a', w \leftarrow b'\}\ (\mathsf{fix}\ (x\ =\ aa' \cdot x\ +\ bb' \cdot x\ )) \\
\sim & \quad \{ \textit{ substitution } \} \\
 & \mathsf{fix}\ (x\ =\ aw.x + bw.x)
\end{array}
$$

Left join:
The behaviour of a left join is a little more complex: before renaming, all computations of $\mathbb{C}$ in which ports i and j are activated independently of each other must be removed. Again this is specified by a new process combinator $\mathsf{force}\,c$ which forces the joint activation of a set c of ports. Formally,

$$\mathsf{force}\,c\ =\ [\!(\alpha_{\mathsf{force}\,c})\!] \tag{5.22}$$

where

$$
\begin{aligned}
\alpha_{\mathsf{force}\,c}\ &=\ \nu \xrightarrow{\omega} \mathcal{P}(\mathbb{A}\times\nu) \xrightarrow{\mathsf{f}_c} \mathcal{P}(\mathbb{A}\times\nu) \\
\mathsf{f}_c\ s\ &= \{\langle a, u\rangle \in s |\ a \cap c\ \subseteq\ \{\emptyset, c\}\}
\end{aligned}
$$

Thus

$$\mathsf{port}.[\![(z <^i_j \mathbb{C})]\!]\ =\ \{z \leftarrow i, z \leftarrow j\}\ \mathsf{force}\,\{i, j\}\,\mathsf{port}.[\![\mathbb{C}]\!]$$

On the other hand, the data flow specification $\mathsf{data}.[\![z <^i_j \mathbb{C}]\!] : \mathbb{D}^n \times U \longleftarrow \mathbb{D}^{m-1} \times U$ is given by

$$t'\ (\mathsf{data}.[\![z <^i_j \mathbb{C}]\!])\ r \quad \text{iff} \quad t'\ (\mathsf{data}.[\![\mathbb{C}]\!])\ t\ \wedge\ r_{|z} = t_{|i,j}\ \wedge\ r_{\#z} = t_{\#i} = t_{\#j}$$

Example 5.3 *A simple, but useful, illustration of this combinator is the* broadcaster *connector depicted in Fig. 5.3. It is obtained by a left join of the source ports of two synchronous channels put in parallel. Its behavioural pattern is computed as*

$$\mathbb{B} \triangleq a <^{c'}_{b'} (\, b' \longrightarrow b \odot c' \longrightarrow c \,) \quad = \quad \begin{tikzcd} & c \\ a \arrow[ru, bend left] \arrow[rd, bend right] & \\ & b \end{tikzcd}$$

Figure 5.3: A *broadcaster*.

follows:

$$\begin{array}{ll}
& \mathsf{port}.[\![\mathsf{B}]\!] \\
\sim & \qquad \{\, \textit{definition of left join and synchronous channel} \} \\
& \{a \leftarrow c', a \leftarrow b'\}\, \mathsf{force}\, \{c', b'\}\, (\mathsf{fix}\ (x \ = \ c'c \cdot x) \odot \mathsf{fix}\ (x \ = \ b'b \cdot x)) \\
\sim & \qquad \{\, \textit{parallel composition and expansion law}\ (4.14) \} \\
& \{a \leftarrow c', a \leftarrow b'\}\, \mathsf{force}\, \{c', b'\}\, (\mathsf{fix}\ (x \ = \ c'c \cdot x + b'b \cdot x + c'cb'b \cdot x)) \\
\sim & \qquad \{\, \textit{definition of}\ \mathsf{force}\ \textit{and substituion} \} \\
& \mathsf{fix}\ (x \ = \ acb \cdot x)
\end{array}$$

Notice that the broadcaster *connector could also be realised replacing $\odot$ by $\otimes$. In that case,* force *would have no effect in the computation of the left join. Notice, by the way, that all these examples can be stated either in terms of $\boxtimes$ and $\boxdot$, orelse, as we did, in terms of their strong versions $\otimes$ and $\odot$.*

5.3 Propagation of context dependent behaviour

The study of context dependent behaviour, and its propagation by composition, in exogenous coordination models was motivated by the behaviour of channels which react differently depending on the presence or absence of information at their ports. The prototypical case is the Reo *lossy* channel defined, in our formalism, by (5.12) and (5.13). In this section we show that the proposed model, with *negative* information, is able the pass two tests which constitute the hallmark of propagation of context dependent behaviour. They are concerned with the behaviour of a *lossy* channel composed either with a synchronous channel or an empty *fifo*$_1$. We formulate then as two lemmas to sustain the claim made about thsi model.

Lemma 5.1 *Whenever a* lossy *channel is composed (via $\odot$ and* hook*) with a syn-*

chronous channel, on either side, the result must be again a lossy *channel,* i.e.

$$(\, a \dashrightarrow a' \;\odot\; b' \longmapsto b \,)\, \Lsh^{b'}_{a'} \;=\; a \dashrightarrow b \tag{5.23}$$

$$(\, a \longmapsto a' \;\odot\; b' \dashrightarrow b \,)\, \Lsh^{b'}_{a'} \;=\; a \dashrightarrow b \tag{5.24}$$

Proof. We concentrate on the behaviour part. For the data component of the definition just observe that the identity relation is always the identity for relational composition. Thus, for (5.23),

$$\begin{aligned}
& \mathsf{port}.[\![(\, a \dashrightarrow a' \;\odot\; b' \longmapsto b \,)\, \Lsh^{b'}_{a'}]\!] \\
\sim \quad & \{ \text{definition of } \odot \text{ and hook} \} \\
& \mathsf{fix}\,(X \;=\; aa' \cdot X + a\tilde{a}' \cdot X + aa'bb' \cdot X + a\tilde{a}'bb' \cdot X + bb' \cdot X + a\tilde{a}'\tilde{b}\tilde{b}' \cdot X) \\
\sim \quad & \{ \text{definition of } \mathsf{hide} \; \} \\
& \mathsf{fix}\,(X \;=\; ab \cdot X + a\tilde{b} \cdot X) \\
\sim \quad & \{ \text{definition of a } \textit{lossy} \text{ channel} \} \\
& \mathsf{port}.[\![\, a \dashrightarrow b \,]\!]
\end{aligned}$$

Similarly, for (5.24),

$$\begin{aligned}
& \mathsf{port}.[\![(\, a \longmapsto a' \;\odot\; b' \dashrightarrow b \,)\, \Lsh^{b'}_{a'}]\!] \\
\sim \quad & \{ \text{definition of } \odot \text{ and hook, expansion law} \} \\
& \mathsf{fix}\,(X \;=\; aa' \cdot X + bb' \cdot X + \tilde{b}b' \cdot X + aa'bb' \cdot X + aa'\tilde{b}b' \cdot X + \tilde{a}\tilde{a}'b\tilde{b}' \cdot X) \\
\sim \quad & \{ \text{definition of } \mathsf{hide} \; \} \\
& \mathsf{fix}\,(X \;=\; ab \cdot X + a\tilde{b} \cdot X + \tilde{a}b \cdot X) \\
\sim \quad & \{ \text{property (5.10), + idempotent} \} \\
& \mathsf{fix}\,(X \;=\; ab \cdot X + a\tilde{b} \cdot X) \\
\sim \quad & \{ \text{definition of a } \textit{lossy} \text{ channel} \} \\
& \mathsf{port}.[\![\, a \dashrightarrow b \,]\!]
\end{aligned}$$

□

Lemma 5.2 *Whenever a* fifo$_1$ *is composed on the right with a* lossy *channel data must flow through the latter to the former as the input port of an empty* fifo$_1$ *is always presenting a reading request. Therefore, no data is lost. Formally,*

$$(\, a \dashrightarrow a' \;\odot\; b' \vdash\!\square\!\!\rightarrow b \,)\, \Lsh^{b'}_{a'} \;=\; a \vdash\!\square\!\!\rightarrow b \tag{5.25}$$

Proof. Let $L = \mathsf{port}.[\![\ a \dashrightarrow a'\ ]\!]$ and $F = \mathsf{port}.[\![\ b' \vdash\!\square\!\!\rightarrow b\ ]\!]$. Then

$$\mathsf{port}.[\![(\ a \dashrightarrow a' \ \odot\ b' \vdash\!\square\!\!\rightarrow b\ )\ \Lsh^{b'}_{a'}]\!]$$

$\sim$ $\{$ definition of channels, $\odot$ and hook, expansion law$\}$

$$aa' \cdot (L \odot F)\ \Lsh^{b'}_{a'} + a\tilde{a}' \cdot (L \odot F)\ \Lsh^{b'}_{a'} + b' \cdot (L \odot b \cdot F)\ \Lsh^{b'}_{a'} +$$
$$aa'b' \cdot (L \odot b \cdot F)\ \Lsh^{b'}_{a'} + a\tilde{a}'b' \cdot (L \odot b \cdot F)\ \Lsh^{b'}_{a'} + a\tilde{a}'\tilde{b}' \cdot (L \odot b \cdot F)\ \Lsh^{b'}_{a'}$$

$\sim$ $\{$ property (5.10), + idempotent$\}$

$$aa' \cdot (L \odot F)\ \Lsh^{b'}_{a'} + a\tilde{a}' \cdot (L \odot F)\ \Lsh^{b'}_{a'} + b' \cdot (L \odot b \cdot F)\ \Lsh^{b'}_{a'} +$$
$$aa'b' \cdot (L \odot b \cdot F)\ \Lsh^{b'}_{a'} + a\tilde{a}'\tilde{b}' \cdot (L \odot b \cdot F)\ \Lsh^{b'}_{a'}$$

$\sim$ $\{$ definition of hide $\}$

$$a \cdot (L \odot b \cdot F)\ \Lsh^{b'}_{a'} + a \cdot (L \odot b \cdot F)\ \Lsh^{b'}_{a'}$$

$\sim$ $\{$ + idempotent$\}$

$$a \cdot (L \odot b \cdot F)\ \Lsh^{b'}_{a'}$$

$\sim$ $\{$ definition of channels, $\odot$ and hook, expansion law$\}$

$$a \cdot (aa' \cdot (L \odot F)\ \Lsh^{b'}_{a'} + a\tilde{a}' \cdot (L \odot F)\ \Lsh^{b'}_{a'} + b \cdot (L \odot F)\ \Lsh^{b'}_{a'} + a\tilde{a}'b \cdot (L \odot F)\ \Lsh^{b'}_{a'} +$$
$$a\tilde{a}'\tilde{b} \cdot (L \odot F)\ \Lsh^{b'}_{a'})$$

$\sim$ $\{$ definition of hide $\}$

$$a \cdot b \cdot (L \odot F)\ \Lsh^{b'}_{a'}$$

$\sim$ $\{$ introducing $\mathsf{fix}$$\}$

$$\mathsf{fix}\ (X\ =\ a \cdot b \cdot X)$$

$\sim$ $\{$ definition of a fifo_1 channel$\}$

$$\mathsf{port}.[\![\ a \vdash\!\square\!\!\rightarrow b\ ]\!]$$

For the data component notice that, once the *lossy* channel never looses any data, as just shown, its static semantics, in this particular composition, is the identity relation. Therefore

$$(\mathsf{data}.[\![\ a \dashrightarrow a' \ \odot\ b' \vdash\!\square\!\!\rightarrow b\ )\ \Lsh^{b'}_{a'}]\!] \quad = \quad \mathsf{data}.[\![\ a \vdash\!\square\!\!\rightarrow b\ ]\!]$$

□

Lemmas 5.1 and 5.2 establish the adequacy of this model to propagate context dependent behaviour. It is also instructive to compute the joint behaviour of a *fifo*$_1$ with a *lossy* channel. This yields, for $F = \mathsf{port}.[\![\ a \vdash\!\square\!\rightarrowtail a'\ ]\!]$ and $L = \mathsf{port}.[\![\ b' \dashrightarrow b\ ]\!]$,

$$
\begin{array}{ll}
& \mathsf{port}.[\![(\ a \vdash\!\square\!\rightarrowtail a' \quad \odot \quad b' \dashrightarrow b\ )\ \Lsh^{b'}_{a'}]\!] \\
\sim & \qquad \{\text{ definition of channels, } \odot \text{ and hook, expansion law}\} \\
& a \cdot (a' \cdot F \odot L)\ \Lsh^{b'}_{a'}\ + bb' \cdot (F \odot L)\ \Lsh^{b'}_{a'}\ + \tilde{b}b' \cdot (F \odot L)\ \Lsh^{b'}_{a'}\ + \\
& abb' \cdot (F \odot L)\ \Lsh^{b'}_{a'}\ + a\tilde{b}b' \cdot (F \odot L)\ \Lsh^{b'}_{a'}\ + \tilde{a}\tilde{b}b' \cdot (F \odot L)\ \Lsh^{b'}_{a'} \\
\sim & \qquad \{\text{ definition of } \mathsf{hide}\ \} \\
& a \cdot (a' \cdot F \odot L)\ \Lsh^{b'}_{a'} \\
\sim & \qquad \{\text{ definition of channels, } \odot \text{ and hook, expansion law}\} \\
& a \cdot (a' \cdot (F \odot L)\ \Lsh^{b'}_{a'}\ + bb' \cdot (a' \cdot F \odot L)\ \Lsh^{b'}_{a'}\ + \tilde{b}b' \cdot (a' \cdot F \odot L)\ \Lsh^{b'}_{a'}\ + \\
& a'bb' \cdot (F \odot L)\ \Lsh^{b'}_{a'}\ + a'\tilde{b}b' \cdot (F \odot L)\ \Lsh^{b'}_{a'}\ + \tilde{a}'bb' \cdot (F \odot L)\ \Lsh^{b'}_{a'}\ + \tilde{a}'\tilde{b}b' \cdot (F \odot L)\ \Lsh^{b'}_{a'}) \\
\sim & \qquad \{\text{ property (5.10) and definition of } \mathsf{hide}\ \} \\
& a \cdot (b \cdot (F \odot L)\ \Lsh^{b'}_{a'}\ + \tilde{b} \cdot (F \odot L)\ \Lsh^{b'}_{a'}\ + b \cdot (F \odot L)\ \Lsh^{b'}_{a'}\ + \tilde{b} \cdot (F \odot L)\ \Lsh^{b'}_{a'}) \\
\sim & \qquad \{ + \text{ idempotent}\} \\
& a \cdot (b \cdot (F \odot L)\ \Lsh^{b'}_{a'}\ + \tilde{b} \cdot (F \odot L)\ \Lsh^{b'}_{a'}) \\
\sim & \qquad \{\text{ introducing } \mathsf{fix}\} \\
& \mathsf{fix}\ (X\ =\ a \cdot (b \cdot X + \tilde{b} \cdot X))
\end{array}
$$

5.4 Towards a connector calculus

Notions of connector equivalence and refinement can be defined, entailing the basis for a connector calculus to reason about the coordination layer of applications. For connectors with identical signatures, refinement corresponds, at the data level, to relational inclusion, as one would expect. In this subsection, however, our attention will be focussed on the behavioural side.

Having above defined behaviours coalgebraically, we get for free the notion of bisimulation associated to functor $\mathcal{P}(\mathbb{A} \times \mathsf{Id})$:

Definition 5.2 *A relation S on processes is a simulation iff*

$$pSq\ \rightarrow\ \forall_{\langle c,p' \rangle \in \omega\, p}\ .\ \exists_{\langle c',q' \rangle \in \omega\, q}\ .\ c = c'\ \wedge\ p'Sq' \tag{5.26}$$

A bisimulation is a simulation whose relational converse is also a simulation. As usual, we denote by $\precsim$ and $\sim$ the similarity and bisimilarity relation, respectively, corresponding to the greatest simulation and bisimulation.

Clearly, by (5.26), one gets

$$\mathsf{port}.[\![\ \bullet \longrightarrow \bullet\ ]\!] \precsim \mathsf{port}.[\![\ \bullet \overset{\cdots}{\dashrightarrow} \bullet\ ]\!]$$

or

$$\mathsf{port}.[\![\ a \longrightarrow b \odot b \longrightarrow c\ ]\!] \sim \mathsf{port}.[\![\ a \longrightarrow c\ ]\!]$$

A richer, weaker, inequational calculus is derived from the fact that actions in $\mathbb{A}$ form a semilattice

$$\langle \mathcal{P}(\mathbb{P} \cup \{\tau\}) \times \mathcal{P}\mathbb{P}, \subseteq \times \subseteq, \langle \emptyset, \emptyset \rangle \rangle$$

by relaxing (5.26) to require $c(\subseteq \times \subseteq)c'$ instead of action equality $c = c'$. Under this new similarity relation, asynchrony appears as a refinement of synchrony as in, *e.g.*,

$$\mathsf{port}.[\![\ \bullet \vdash\!\square\!\!\rightarrow \bullet\ ]\!] \precsim \mathsf{port}.[\![\ \bullet \longrightarrow \bullet\ ]\!]$$

We shall now consider, in some detail, a number of examples of connector composition with a focus on computing the emergent behaviour. This is expected to provide a 'flavour' of the model in action.

Example 5.4 *The first example is concerned with the proof of the following fact on the '*impossible*' (or deadlocked) connector depicted in Fig. 5.4. In particular, this example indicates our semantics is compliant with the operational description of* Reo *in [Arb04]:*

$$\mathsf{port}.[\![\mathsf{Imp}]\!] \sim \mathbf{0} \tag{5.27}$$

Actually, Imp *is built as*

$$\mathsf{Imp} \triangleq a <^a_d b <^b_e (\, d \vdash\!\!\!\relbar\!\!\!\relbar\!\!\dashv e \odot (\, a \longrightarrow a' \boxplus b \longrightarrow b'\,)\,{}^{a'}_{b'} > w))$$

Therefore

$$\begin{array}{ll}
 & \mathsf{port}.[\![\mathsf{Imp}]\!] \\
\sim & \quad \{\ \textit{definiton of a left join, } \sigma = \{a \leftarrow d, b \leftarrow e\}\} \\
 & \sigma\ \mathsf{force}\,\{a, b, d, e\}\,(\mathsf{fix}\ (X = de \cdot X) \odot \mathsf{fix}\ (X = aw \cdot X + bw \cdot X)) \\
\sim & \quad \{\ \textit{expansion law } (4.14)\} \\
 & \sigma\ \mathsf{force}\,\{a, b, d, e\}\,(\mathsf{fix}\ (X = deaw \cdot X + debw \cdot X + de \cdot X + aw \cdot X + bw \cdot X)) \\
\sim & \quad \{\ \textit{definition of}\,\mathsf{force}\} \\
 & \mathbf{0}
\end{array}$$

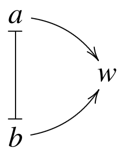


Figure 5.4: Imp — An '*impossible*' connector.

Example 5.5 *As a second example, consider the implementation of an* asynchronous drain *by plugging the output port of a* merger *to both input ports of a* synchronous drain. *Formally,*

$$(((\, a \longrightarrow a' \ \boxplus\ b \longrightarrow b' \,)^{b'}_{c'} > w) \ \boxdot\ (\, c \vdash\!\!\!-\!\!\!-\!\!\!\dashv c' \,) z <^{c}_{c'}) \Lsh^{z}_{w} \equiv (\, a \overset{\nabla}{\vdash\!\!\!-\!\!\!-\!\!\!\dashv} a' \,)$$

$$\mathsf{port}.[\![(((\, a \longrightarrow a' \ \boxplus\ b \longrightarrow b' \,)^{b'}_{c'} > w) \ \boxdot\ z <^{c}_{c'} (\, c \vdash\!\!\!-\!\!\!-\!\!\!\dashv c' \,)) \Lsh^{z}_{w}]\!]$$

$\sim$ { *the* merger *connector in example 5.2 and definition of* drain}

$$(\mathsf{fix}\ (X \ =\ aw \cdot x + bw \cdot X) \boxdot \mathsf{port}.[\![z <^{c}_{c'} (\, c \vdash\!\!\!-\!\!\!-\!\!\!\dashv c' \,)) \Lsh^{z}_{w}]\!]$$

$\sim$ { *definition of left join*}

$$(\mathsf{fix}\ (X \ =\ aw \cdot X + bw \cdot X) \boxdot \{z \leftarrow c, z \leftarrow c'\}\mathsf{force}\{c, c'\}\mathsf{fix}\ (X \ =\ cc' \cdot X)) \Lsh^{z}_{w}$$

$\sim$ { *definition of* force , *substitution*}

$$(\mathsf{fix}\ (X \ =\ aw \cdot X + bw \cdot X) \boxdot \mathsf{fix}\ (X \ =\ z \cdot X)) \Lsh^{z}_{w}$$

$\sim$ { *definition of hook, expansion law*}

$$\mathsf{hide}\{z, w\}\ \mathsf{fix}\ (X \ =\ awz \cdot X + bwz \cdot X + aw \cdot X + bw \cdot X + z \cdot X)$$

$\sim$ { *definition of* hide }

$$\mathsf{fix}\ (X \ =\ a \cdot X + b \cdot X$$

$\sim$ { *behaviour of an* asynchronous drain}

$$\mathsf{port}.[\![\ a \overset{\nabla}{\vdash\!\!\!-\!\!\!-\!\!\!\dashv} b\ ]\!]$$

Example 5.6 *Consider now the construction of a* synchronization barrier *as depicted in Fig. 5.5. A synchronization barrier connector enables data items to pass from a to b and from c to d, but only in a synchronised way. A possible way of building* SB *is as follows (the use of* ⊡ *instead of* ⊠ *leads to the same result):*

$$((\, a \longrightarrow a' \ \boxtimes\ c \longrightarrow c' \,) \boxtimes (y <^{e'}_{d'} (x <^{b'}_{e} (\, b \longrightarrow b' \ \boxtimes\ e \vdash\!\!\!-\!\!\!-\!\!\!\dashv e' \,) \boxtimes\ d \longrightarrow d' \,)) \Lsh^{x,y}_{a',c'}$$

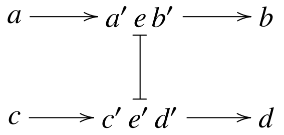


Figure 5.5: SB — A *synchronization barrier.*

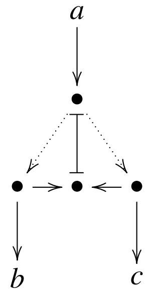


Figure 5.6: XR — The *exclusive router* connector.

Its behaviour is computed as

$$
\begin{array}{ll}
 & \mathsf{port}.[\![\mathsf{SB}]\!] \\
\sim & \quad \{\ \textit{definition of}\ \boxtimes\} \\
 & (\mathsf{fix}\ (X\ =\ aa'cc' \bullet X) \boxtimes (y <^{e'}_{d'} (x <^{b'}_{e} \mathsf{fix}\ (X\ =\ bb'ee'dd' \bullet X)))\ \Lsh^{x,y}_{a',c'} \\
\sim & \quad \{\ \textit{definition of}\ \text{left join}\} \\
 & (\mathsf{fix}\ (X\ =\ aa'cc' \bullet X) \boxtimes \mathsf{fix}\ (X\ =\ xbdy \bullet X)))\ \Lsh^{x,y}_{a',c'} \\
\sim & \quad \{\ \textit{definition of}\ \text{hook}\ \textit{and}\ \boxtimes\} \\
 & \mathsf{fix}\ (X\ =\ acbd \bullet X)
\end{array}
$$

Again the use of normal or strong versions is irrelevant in this example.

Example 5.7 *As a final example, let us compute the behaviour of* XR*, the* exclusive router *connector depicted in Fig. 5.6. The intended behaviour for this connector is to transmit either in b or c, but not in both, whatever receives in a.*

One component of XR*, depicted in the lower part of the diagram, is the* right join *by e′ and d′ mapping to new port z, of two* broadcasters *composed by* ⊠*. Each broadcaster is obtained by left joining the relevant synchronous channels. The as-*

sembly process of XR_1 *is represented as*

$$\begin{tikzcd} b'e \arrow[r] \arrow[d] & e'd' & dc' \arrow[l] \arrow[d] & \arrow[r, squiggly] & w_1 \arrow[r] \arrow[d] & z & w_2 \arrow[l] \arrow[d] \\ b & & c & & b & & c \end{tikzcd}$$

The computed behavioural pattern is

$$\mathsf{port}.[\![\mathsf{XR}_1]\!] \;\sim\; \mathsf{fix}\,(X \;=\; bzw_1 \cdot X + czw_2 \cdot X)$$

The other component, XR_2 *is a left join of two lossy channels and a drain, mapping their source ports, h, g and f, to w, sequentially composed with a synchronous channel from a to a′,* i.e.,

$$\begin{tikzcd} & a \arrow[d] & \\ & w \arrow[dl, dotted] \arrow[d, dash] \arrow[dr, dotted] & \\ h' & f' & g' \end{tikzcd}$$

Its behavioural pattern is computed as follows:

$$\begin{aligned}
& \mathsf{port}.[\![\mathsf{XR}_2]\!] \\
\sim \quad & \{\ \textit{definiton of}\ \mathsf{XR}_2\} \\
& \mathsf{port}.[\![(\ a \longrightarrow a\ \odot w <^f_r\ r <^h_g (\ h \dashrightarrow h'\ \otimes\ f \longmapsto\!\dashv f'\ \otimes\ g \dashrightarrow g'\ ))\ \lrcorner^{w}_{a'}]\!] \\
\sim \quad & \{\ \textit{definitions of channels}, \otimes, \textit{left join and hook; expansion law}\ (4.14)\} \\
& \mathsf{hide}\{a', w\}\,(\mathsf{fix}\,(X \;=\; aa' \cdot X)\ \odot \\
& \qquad\qquad \mathsf{fix}\,(X \;=\; wh'f'g' \cdot X + w\widetilde{h'}f'g' \cdot X + wh'f'\widetilde{g'} \cdot X + wf'\widetilde{h'}\widetilde{g'} \cdot X)) \\
\sim \quad & \{\ \textit{definition of}\ \odot\} \\
& \mathsf{hide}\{a', w\}\,\mathsf{fix}\,(X \;=\; aa' \cdot X + wh'f'g' \cdot X + w\widetilde{h'}f'g' \cdot X + wh'f'\widetilde{g'} \cdot X + wf'\widetilde{h'}\widetilde{g'} \cdot X \\
& \qquad + aa'wh'f'g' \cdot X + aa'w\widetilde{h'}f'g' \cdot X + aa'wh'f'\widetilde{g'} \cdot X + aa'wf'\widetilde{h'}\widetilde{g'} \cdot X \\
& \qquad + \widetilde{a}\widetilde{a'}wh'f'g' \cdot X + \widetilde{a}\widetilde{a'}w\widetilde{h'}f'g' \cdot X + \widetilde{a}\widetilde{a'}wh'f'\widetilde{g'} \cdot X + \widetilde{a}\widetilde{a'}wf'\widetilde{h'}\widetilde{g'} \cdot X) \\
\sim \quad & \{\ \textit{definition of}\ \mathsf{hide}\} \\
& \mathsf{fix}\,(X \;=\; ah'f'g' \cdot X + a\widetilde{h'}f'g' \cdot X + ah'f'\widetilde{g'} \cdot X + af'\widetilde{h'}\widetilde{g'} \cdot X)
\end{aligned}$$

Finally, the connector XR *is assembled as*

$$\mathsf{XR} \;\triangleq\; (\mathsf{XR}_1 \otimes \mathsf{XR}_2)\ \lrcorner^{w_1, w_2, f'}_{h', g', w} \tag{5.28}$$

leading, as expected, to

$$
\begin{array}{ll}
 & \mathsf{port}.[\![\mathsf{XR}]\!] \\
\sim & \quad \{\, \textit{by } (5.28) \} \\
 & \mathsf{hide}\{h', g', w, w_1, w_2, f'\}\, (\mathsf{port}.[\![\mathsf{XR}_1]\!] \otimes \mathsf{port}.[\![\mathsf{XR}_2]\!]) \\
\sim & \quad \{\, \textit{computed above} \} \\
 & \mathsf{hide}\{h', g', w, w_1, w_2, f'\}\, (\mathsf{fix}\ (X \ = \ bzw_1 \cdot X + czw_2 \cdot X) \ \otimes \\
 & \qquad\qquad \mathsf{fix}\ (X \ = \ ah'f'g' \cdot X + a\widetilde{h'}f'g' \cdot X + ah'f'\widetilde{g'} \cdot X + af'\widetilde{h'}\widetilde{g'} \cdot X)) \\
\sim & \quad \{\, \textit{definition of} \otimes\textit{; expansion law} \} \\
 & \mathsf{hide}\{h', g', w, w_1, w_2, f'\} \\
 & \quad \mathsf{fix}\,(X \ = \ bzw_1ah'f'g' \cdot X + bzw_1a\widetilde{h'}f'g' \cdot X + bzw_1ah'f'\widetilde{g'} \cdot X + bzw_1af'\widetilde{h'}\widetilde{g'} \cdot X + \\
 & \qquad\qquad czw_2ah'f'g' \cdot X + czw_2a\widetilde{h'}f'g' \cdot X + czw_2ah'f'\widetilde{g'} \cdot X + czw_2af'\widetilde{h'}\widetilde{g'} \cdot X + \\
 & \qquad\qquad \widetilde{b}\widetilde{z}\widetilde{w_1}a\widetilde{h'}f'g' \cdot X + \widetilde{b}\widetilde{z}\widetilde{w_1}ah'f'\widetilde{g'} \cdot X + \widetilde{b}\widetilde{z}\widetilde{w_1}af'\widetilde{h'}\widetilde{g'} \cdot X + \\
 & \qquad\qquad \widetilde{c}\widetilde{z}\widetilde{w_2}a\widetilde{h'}f'g' \cdot X + \widetilde{c}\widetilde{z}\widetilde{w_2}ah'f'\widetilde{g'} \cdot X + \widetilde{c}\widetilde{z}\widetilde{w_2}af'\widetilde{h'}\widetilde{g'} \cdot X) \\
\sim & \quad \{\, \textit{definition of } \mathsf{hide} \} \\
 & \mathsf{fix}\,(X \ = \ ab \cdot X + ab \cdot X + ac \cdot X + ac \cdot X + a\widetilde{b} \cdot X + a\widetilde{b} \cdot X + a\widetilde{c} \cdot X + a\widetilde{c} \cdot X) \\
\sim & \quad \{\, + \textit{ idempotent} \} \\
 & \mathsf{fix}\ (X \ = \ ab \cdot X + ab \cdot X + a\widetilde{b} \cdot X + a\widetilde{c} \cdot X)
\end{array}
$$

Notice that the application of hide *in the calculation above made use of possibility of keeping terms where the intersection of their prefix set of ports with the argument of* hide *reduce to a* negated output *port (cases of $\widetilde{h'}$ and $\widetilde{g'}$).*

5.5 Configurations

Chapter 4 characterised interfaces for components given by process terms over the collection of (input and output) *ports*, assuming that the active entities inside the service consume or make available data items. Such a process term represents the service *bussiness protocol*, its externally perceived behaviour referred to as the service *use pattern*. As one could expect, such *use patterns* are defined in the same process language used for specifying connectors in the previous sections.

Having such a characterisation for components and a model for the coordination layer, we can now move to *the whole picture*. Technically, this will be called a *configuration*.

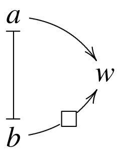


Figure 5.7: An *alternate merger*.

A *configuration*, like the one depicted in Fig. 5.1 is simply a collection of services, characterized by their interfaces, interconnected through an *orchestrator*, *i.e.*, a connector built from elementary connectors using the combinators introduced above. Actually, we have now all the ingredients to replace the empty circle in that figure by a suitable connector which enforces strict alternation of data sources. Such is the purpose of the *alternate merger* depicted in Fig. 6.9. Formally, $\mathbb{AM}$ is defined as

$$b <^{d'}_{f} a <^{d}_{c} (\ c \longrightarrow c' \ \odot \ d \longmapsto\!\dashv d' \ \odot \ f \ -\!\square\!\!\rightarrow f')^{c'}_{f'} > w$$

Following the method illustrated in the previous section, its behavioural pattern is easily computed from this expression:

$$\mathsf{port}.[\![\mathbb{AM}]\!] \;=\; \mathsf{fix}\,(x \;=\; abw.w.x) \tag{5.29}$$

Formally,

Definition 5.3 *A configuration involving a collection $S = \{S_i |\ i \in n\}$ of components is a tuple*

$$\langle U, \mathbb{C}, \sigma \rangle \tag{5.30}$$

where

$$U = use(S_1) \odot use(S_2) \odot \cdots \odot use(S_n) \tag{5.31}$$

is the (joint) use pattern for S, $\mathbb{C}$ is a connector and σ a mapping of ports in S to ports in $\mathbb{C}$.

The role of renaming σ in the definition above is to syntactically enforce a link between a service port and a connector end. Clearly, σ respects polarities: output (respectively, input) service ports can only be connected to connectors source (respectively, sink) ends. Interaction is achieved by the simultaneous activation of identically named ports.

Actually, the relevant point concerning configurations is the semantics of interaction between the *connector's behavioural pattern* and the *joint use patterns* of the involved services. This is captured by a synchronous product $\otimes$ for a quite peculiar θ, which is expected to capture the requirements below. Recall that, at each point in the execution of a configuration, θ 'decides' the result of the interaction combining a set of ports offered by the services' side and the sets of positive and negative connector's ends. Therefore,

- There is no interaction if the connector requires absence of port requests in an end linked to a port activated by a service.
- Similarly, there is no interaction if the connector's side offers free ports (*i.e.*, ports that are not connected to services).
- The dual situation is allowed, *i.e.*, if the services' side offer activation of all ports plugged to the ones offered by the connector, their intersection is the resulting interaction.
- Finally, free ports on the service side (*i.e.*, ports that are not connected to a connector's end) are not affected by θ: their activation depends only on the service they belong to.

Formally, this is captured in the following definition.

Definition 5.4 *The behaviour $bh(\Gamma)$ of a configuration $\Gamma = \langle U, \mathbb{C}, \sigma\rangle$ is given by*

$$bh(\Gamma) \;=\; \sigma\, U \;\otimes\; \mathsf{port}.[\![\mathbb{C}]\!] \tag{5.32}$$

where θ underlying the $\otimes$ connective is given by

$$s\;\theta\;\langle p, np\rangle \;=\; \begin{cases} (s \cup \{\tau\}) \cap (p \;\cup\; \mathsf{free}) & \Leftarrow\; s \cap np = \emptyset \;\wedge\; p \subseteq (s \cup \{\tau\}) \\ s \,\cap\, \mathsf{free} & \Leftarrow\; otherwise \end{cases} \tag{5.33}$$

where free *denotes the set of unplugged ports in U,* i.e.*, not in the domain of mapping σ.*

Note that in the above definition θ relates different types of actions, its signature being $\theta : \mathcal{P}\mathbb{P} \times (\mathcal{P}(\mathbb{P} \cup \{\tau\}) \times \mathcal{P}\mathbb{P}) \longrightarrow (\mathcal{P}\mathbb{P} \cup \{\tau\})$. This means the behaviour of a configuration is expressed in terms of services' ports and the unobservable action τ (which, as explained on introducing the *hook* combinator, can not be ignored). It also means that the resulting synchronous product is not commutative, which however does not restrict the expressive power of this approach.

5.5.1 Examples

In the sequel the use of configurations, and the computation of their behaviours, is illustrated by a few examples.

Example 5.8 *Consider an elementary* banking system *composed by an ATM machine, a Bank, and a DBRep service whose purpose is to backup all the messages flowing through the connector. Therefore, all messages are replicated before being stored. Configuration BS, depicted in Fig. 5.8, is specified as*

$$BS \;=\; \langle WBS,\, \mathbb{CON},\, \sigma_{BS}\rangle$$

where

$$\begin{aligned} WBS &= use(ATM) \odot use(Bank) \odot use(DBRep) \\ \sigma_{HS} &= \{a \leftarrow A_{rq}, e \leftarrow A_{rs}, c \leftarrow DB_r, f \leftarrow DB_p, d \leftarrow B_{rs}, b \leftarrow B_{rq}\} \end{aligned}$$

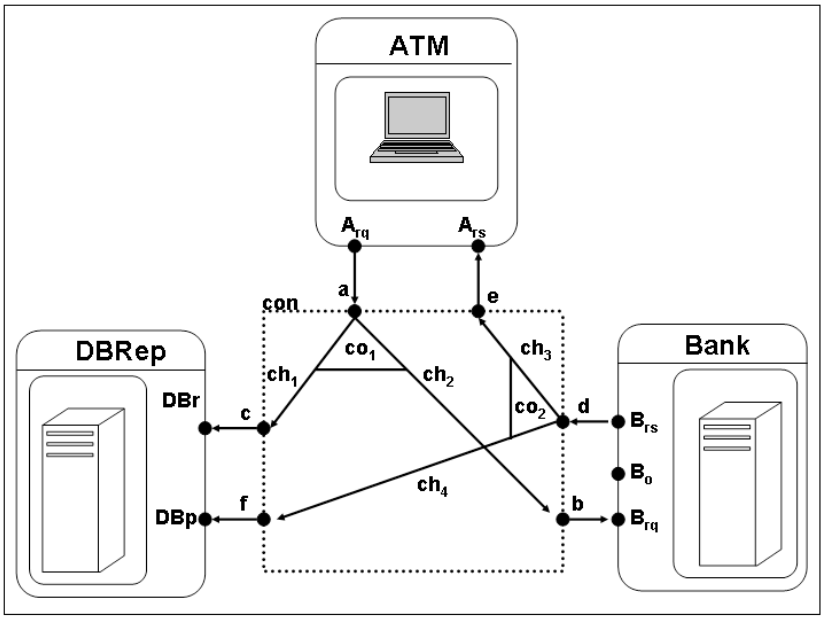


Figure 5.8: A elementary bank system

Consider the following use patterns of each web service after port renaming by

σ_{BS}:

$$\begin{aligned}
use(ATM) &= \mathsf{fix}\,(X = a \cdot e \cdot X)\\
use(Bank) &= \mathsf{fix}\,(Y = b \cdot d \cdot Y)\\
use(DBRep) &= \mathsf{fix}\,(Z = c \cdot Z + f \cdot Z + cf \cdot Z)
\end{aligned}$$

Connector $\mathbb{CON}$ *behaves like a double broadcaster (hence its name). Its behaviour allows for both the simultaneous or independent activation of each broadcast (*co_1 *or* co_2*) as shown by the following computation:*

$$\begin{aligned}
&\mathsf{port}.[\![ch_1]\!] \;=\; \mathsf{fix}\,(X = b'b \cdot X),\ \mathsf{port}.[\![ch_2]\!] \;=\; \mathsf{fix}\,(X = c'c \cdot X)\\
&\mathsf{port}.[\![co_1]\!] \;=\; \mathsf{port}.[\![(a <^{b'}_{c'} (ch_1 \odot ch_2))]\!] \;=\; \mathsf{fix}\,(X = abc \cdot X)\\
&\mathsf{port}.[\![ch_3]\!] \;=\; \mathsf{fix}\,(x = e'e \cdot X),\ \mathsf{port}.[\![ch_4]\!] \;=\; \mathsf{fix}\,(x = f'f \cdot X)\\
&\mathsf{port}.[\![co_2]\!] \;=\; \mathsf{port}.[\![(d <^{e'}_{f'} (ch_3 \odot ch_4))]\!] \;=\; \mathsf{fix}\,(X = def \cdot X)\\
&\mathsf{port}.[\![\mathbb{CON}]\!] \;=\; \mathsf{port}.[\![(co_1 \odot co_2)]\!] \;=\; \mathsf{fix}\,(X = abc \cdot X + def \cdot X + abcdef \cdot X)
\end{aligned}$$

To determine $bh(BS)$ *one needs to expand* WBS*. According to* (5.33)*, we need only to look for summands prefixed by sets of ports which are super-sets of prefix sets in* $\mathsf{port}.[\![\mathbb{CON}]\!]$. *For the first level of expansion alternative* $abc\cdot(e\cdot X \boxdot d\cdot Y \boxdot Z)$ *is the only one to* θ*-compose with abc in* $\mathsf{port}.[\![\mathbb{CON}]\!]$*, resulting in abc again. Then, consider the expansion of term* $(e\cdot X \odot d\cdot Y \odot z)$*: the only alternative worth to consider (*i.e.*, which does not lead to* 0 *on* θ*-composition) is* $edf \cdot (X \odot Y \odot Z)$*, the resulting interaction being* edf*. From this point on the same expansion pattern repeats. This means that* $bh(BS)$ *becomes:*

$$bh(BS) = \mathsf{fix}\,(X = abc \cdot edf \cdot X) \tag{5.34}$$

Notice how the particular use patterns in the web services act as a constraint *over the admissible behaviour of connector* $\mathbb{CON}$.

This example may be also used to check how definition (5.33) *deals with the presence of unplugged ports, such us port* B_o *in service Bank. Consider, then, the following two alternatives for the use pattern of service Bank:*

$$use(Bank) = \mathsf{fix}\,(Y = bB_o \cdot d \cdot Y) \tag{5.35}$$

$$use(Bank) = \mathsf{fix}\,(Y = b \cdot d \cdot Y + B_o \cdot Y) \tag{5.36}$$

In the expansion of WBS*, expression* (5.35)*, which captures the simultaneous activation of ports b and* B_o*, leads to term* $abB_oc\cdot(e\cdot X \odot d\cdot Y \odot z)$ *which, as* $\mathsf{free} = \{B_o\}$*, entails*

$$bh(BS) = \mathsf{fix}\,(x = abcB_o \cdot edf \cdot X) \tag{5.37}$$

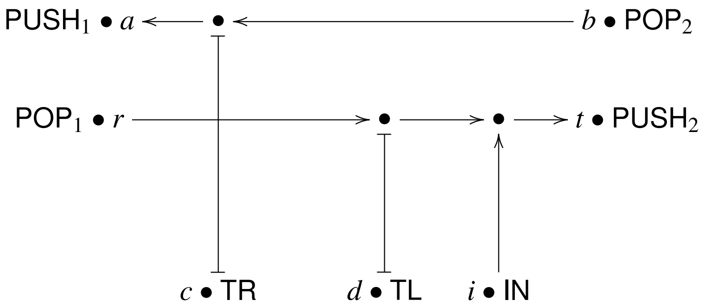


Figure 5.9: A *folder* from two stacks.

Alternative (5.36) *specifies that ports b and B_o are activated in alternative: no term with both b and B_o will appear in the expansion and, therefore, bh*(BS) *remains as given by equation* (5.34).

Example 5.9 *For a second example suppose a configuration with two instances of a service modelling a* stack *and responding on ports* PUSH *and* POP*. A third service intends to build a (electronic version of a paper)* folder *by providing ports to* turn a page left *(*TL*),* turn a page right *(*TR*) and* insert a new page *(*IN*) in the (right side of the) folder. The problem is to orchestrate the three services in such a way that each service* stack *will manage one of the page piles of the folder.*

In [BO03] a solution is given, in the context of a component-based framework, two components modelling stacks are composed by specific operators to implement an interface for a folder. We sketch here an exogenous coordination solution in which the three services do not interact directly, the emergent behaviour of their orchestration being enabled by a suitable connector $\mathbb{SF}$*. The configuration is depicted in Fig. 5.9. It is a small exercise to compute the behavioural pattern associated to* $\mathbb{SF}$*:*

$$\mathsf{port}.[\![\mathbb{SF}]\!] = \mathsf{fix}\,(X = abc \cdot X + rdt \cdot X + it \cdot X + acbrdt \cdot X + acbit \cdot X)$$

Assuming the three services involved can activate each of their ports at any time, the global use pattern of the service layer becomes

$$use(Stack) \otimes use(FInterface) \otimes use(Stack) =$$
$$\mathsf{fix}\,(X = \mathsf{PUSH}_1 \cdot X + \mathsf{POP}_1 \cdot X + \mathsf{TL} \cdot X + \mathsf{TR} \cdot X + \mathsf{IN} \cdot X + \mathsf{PUSH}_2 \cdot X + \mathsf{POP}_2 \cdot X)$$

Finally, the behaviour of the configuration is computed according to definition 3, yielding

$$\mathsf{fix}\,(X = \{\mathsf{TL}, \mathsf{PUSH}_1, \mathsf{POP}_2\} \cdot X + \{\mathsf{TR}, \mathsf{PUSH}_2, \mathsf{POP}_1\} \cdot X + \{\mathsf{IN}, \mathsf{PUSH}_2\} \cdot X)$$

where the port mapping is the one represented in Fig. 5.9. The emergent behaviour of this configuration includes the simultaneous activation of TL, PUSH_1 *and* POP_2, *'implementing' a* turn left *action in the folder interface by a synchronisation between a* pop *in the right stack and a* push *in the left one. And dually for a* turn right. *Also notice port* PUSH_2 *is used either for achieving a* turn right *in the folder or for inserting a new page. None of the three services involved interact directly with one another and even do not need to be aware of each other existence.*

Our last example is part of an attempt to apply exogenous coordination to *interactive systems*, documented in publication [BBC07]. Such systems can be regarded as a special case of the more general class of reactive systems, but presenting some specific characteristics which cannot be ignored in modelling [HT90]. One major issue, for example, is the need to consider interaction with the user, and not only between components of the user interface.

We will discuss here a configuration for a particular situation in an elementary air traffic control system borrowed from [CH08]. The area is, however, vast and challenging. This motivated further research on the intersection of interactive systems modelling and exogenous coordination, whose result is reported in chapter 6.

Example 5.10 *Let us consider a simple air traffic control system. Our example (see Fig. 5.10) is centred on a scenario where aircrafts* A_2 *and* A_3 *are on their final approach to the runway, aircraft* A_1 *is on the runway waiting the response for its 'accepted' to take off requirement, and the tower T is responsible for air traffic control. Aircraft* A_2 *and* A_3 *are on their "downwind leg" and are to be turned onto a heading towards the runway. Before* A_2 *can be turned it must reduce speed. This means that* A_3 *must reduce speed also to avoid loss of separation with* A_2*. Of course,* A_2 *will be allowed to land just after* A_1 *has taken off.*

At this stage we are mainly interested in investigating how to combine interactors in different ways for different scenarios. Investigating the appropriateness of each configuration would be the next step in the design process.

First we express the expected behaviour of the interactors involved in this configuration.

interactor: A_i
ports: $\mathsf{slow}'_\mathsf{i}, \mathsf{turn}'_\mathsf{i}, \mathsf{accept}'_\mathsf{i}$
external behaviour:
$use(A_i) \;=\; \mathsf{fix}\,(X \;=\; \mathsf{slow}'_\mathsf{i} \cdot X \;+\; \mathsf{turn}'_\mathsf{i} \cdot X \;+\; \mathsf{accept}'_\mathsf{i} \cdot X)$, *where* $0 < i \leq 3$.

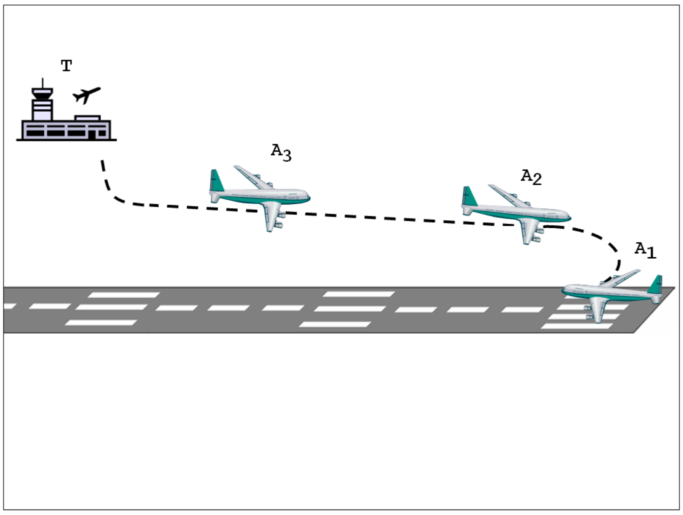


Figure 5.10: Air traffic control configuration

Such a specification represents the three aircrafts involved in the scenario. Each aircraft has three input ports (distinguished by the symbol: ′) available for communication in a non-deterministic manner. The tower is represented by interactor T.

interactor: T
ports: $\mathsf{slow_i}, \mathsf{turn_i}, \mathsf{accept_i}$
external behaviour:
$use(T) \;=\; \mathsf{fix}\,(X \;=\; \mathsf{slow_i} \cdot X \;+\; \mathsf{turn_i} \cdot X \;+\; \mathsf{accept_i} \cdot X)$, *where* $0 < i \leq 3$.

Once the interactors defined, the following step is to define how they will cooperate, i.e.*, we need to represent how the whole system will behave. Such is done by creating an architecture of interactors and connectors.*

The scenario captured by Fig. 5.10 represents a critical situation where the aircrafts must respond to actions appropriately or the safety will be dangerously compromised. So, let us consider a situation where T sends a message $\mathsf{accepted_1}$ *to* A_1*, in order for* A_1 *to take off, the message* $\mathsf{slow_2}$ *to* A_2*, in for* A_2 *to slow before turning to the runway, and the message* $\mathsf{slow_3}$ *to* A_3 *in order for* A_3 *decrease speed maintaining a safety distance to* A_2*. In order to ensure that the response to these actions will happens synchronously we may consider a special connector, called* synchronization barrier *(SB) which enforces that all messages are delivered to their destinations in a synchronous way.*

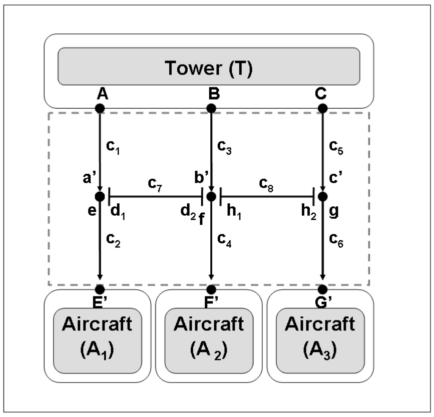


Figure 5.11: Air traffic control configuration - connector

Such a connector (see Fig. 5.11) is an aggregation among six synchronous channels ($c_1, \ldots, c_6$) and two synchronous drains (c_7 and c_8) which are composed using hook and join combinators. This connector is computed starting from the behaviours of the elementary connectors, e.g., $\mathsf{port}.[\![c_1]\!] = \mathsf{fix}\ (x = aa'.x)$*, till the behaviour of the whole connector is calculated:* $\mathsf{port}.[\![SB]\!] = \mathsf{fix}\ (X = abce'f'g' \bullet X)$

The resulting behaviour of this connector means that the six ports must be activated synchronously. It should be noted that, since we are not considering timing issues at this stage, this synchronicity does not meant that the ports are activated concurrently. In the current context, what we are stating is that if one port is activated, then all the other must be activated, before the connector can engage in a new interaction.

The configuration of such a scenario is given by

$$
\begin{aligned}
C_{f_1} &= \langle USC, \mathbb{C}, \sigma_{SC} \rangle, \quad \textit{where} \\
USC &= use(T) \odot use(A_1) \odot use(A_2) \boxdot use(A_2) \\
\mathbb{C} &= SB \\
\sigma_{cf_1} &= \{a \leftarrow A, b \leftarrow B, c \leftarrow C, e' \leftarrow E', f' \leftarrow F', g' \leftarrow G'\}
\end{aligned}
$$

where $A \overset{\text{abv}}{=} \mathsf{accept}_1$, $B = \mathsf{slow}_2$, $C = \mathsf{slow}_3$, $E' = \mathsf{accept}'_1$, $F' = \mathsf{slow}'_2$, *and* $G' = \mathsf{slow}'_3$.

The result of the $\otimes$ *composition of* USC *and* SB *is the behaviour of configuration* C_{f_1}*. There is no need, however, to compute the complete expansion of the* *parallel composition in USC expression, which is*

$$\begin{aligned}\mathsf{fix}\,(X\ =\ & a \bullet X + \cdots + e'. \bullet X + f' \bullet X + g' \bullet X + \\ & ae' \bullet X + \cdots + be' \bullet X + \cdots + ce' \bullet X + \cdots + abce' \bullet X + \cdots + \\ & ae'f' \bullet X + \cdots + be'f' \bullet X + \cdots + ce'f' \bullet X + \cdots + abce'f' \bullet X + \cdots + \\ & ae'f'g' \bullet X + \cdots + be'f'g' \bullet X + \cdots + ce'f'g' \bullet X + \cdots + \underline{abce'f'g'} \bullet X + \cdots + \\ & e'f' \bullet X + e'g' \bullet X + f'g' \bullet X + e'f'g' \bullet X)\end{aligned}$$

because, according to interaction discipline (5.33)*, the only successful case of composition with* $\mathsf{port}.[\![SB]\!]$ *corresponds to the underlined alternative in the expression above. Clearly, the θ-composition of $abce'f'g'$ with $abce'f'g'$ (from the connector side) is $abce'f'g'$, while for all other cases it results in the empty set $\emptyset$. Therefore, and finally,*

$$bh(C_{f_1}) \ = \ \mathsf{fix}\,(X \ = \ abce'f'g' \bullet X)$$

Chapter 6

Case-study: Interactors

Summary

Although presented with a variety of 'flavours', the notion of an interactor*, as an abstract characterisation of an interactive component, is well-known in the area of formal modelling techniques for interactive systems. This chapter proposes to replace traditional, hierarchical, 'tree-like' composition of interactors in the specification of complex interactive systems, by their* exogenous coordination *through general-purpose software* connectors *which assure the flow of data and the meet of synchronisation constraints. The chapter's technical contribution is twofold. First a modal logic is defined to express behavioural properties of both interactors and connectors. The logic is new in the sense that its modalities are indexed by fragments of* sets *of actions to cater for action co-occurrence. Then, this logic is used in the specification of both interactors and coordination layers which orchestrate their interconnection, providing a case-study in the use of software connectors for program coordination. This chapter is based on [BBC09].*

6.1 Introduction

Modern interactive systems resort to increasingly complex architectures of user interface components. With the generalisation of ubiquitous computing, the notion of interactive system itself changed. Single interactive devices have been replaced by frameworks where devices are combined to provide services to a number of different, often anonymous, users accessing them in a competing way. This may explain the increasing interest on rigorous methodologies to develop useful, workable models of such systems. In such a setting, the concept of an *interactor* was originally proposed by Faconti and Paternò [FP90], as an abstraction for a graphical object

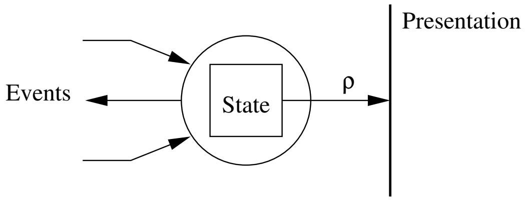


Figure 6.1: A model-based Interactor.

capable of both input and output, typically specified in a process algebra. This was further generalised by Duke and Harrison [DH93] for modelling interactive systems. Interactors become able not only to communicate through i/o ports, but also to convey information about their state through a rendering relation that maps the latter to some presentation medium.

The framework outlined in [DH93], however, does not prescribe a specification notation for the description of interactor state and behaviour. Several possibilities have been considered. One of them, which directly inspired this piece of research, was developed by the third author in [CH01] and resorts to Modal Action Logic (MAL) [RFM91] to specify behavioural constraints. Another one [Pat95] uses LOTOS to express a relation between input and output ports. Actually, the notion of an interactor as a structuring mechanism for formal models of interactive systems, has been an influential one. It has been used, for example, with LOTOS [FP90, Mar95], Lustre [dSDR98], Petri nets [BNP03], Higher Order Processes [DF05], or Modal Action Logic [CH08].

Whatever the approach, modelling complex interactive systems entails creating architectures of interconnected interactors. In [CH01] such models are built hierarchically through 'tree-like' aggregation. Composition is typically achieved by the introduction of additional axioms and/or dedicated interactors to express the control logic for communication. This, in turn, adds dramatically to the complexity of the proposed models. Moreover, it does not promote a clear separation of concerns between modelling interactors and the specification of how they interact with each other.

This chapter, however, adopts an *exogenous coordination* approach to the composition of interactors which entails an effective separation of concerns between the latter and the specification of how they are organised into specific architectures and interact to achieve common goals. Exogenous coordination draws a clear distinction between the *loci* of computational effort and that of interaction control, the latter being blind with respect to the structure of values and procedures which typically depend on the application domains.

Research reported here is a follow-up of a previous attempt to use the coordination paradigm to express the logic governing the composition of interactors, reported in [BBC07], where a process algebra framework was used to specify connector's behavioural constraints. This, however, proved difficult to smoothly combine with interactors whose evolution is typically given by *modal logic* assertions.

Therefore, in this chapter an extension to Hennessy-Milner logic [HM85] is proposed to express behavioural properties of both interactors and connectors. The novelty in the logic is the fact that its modalities are indexed by *sets* of actions to cater for action co-occurrence. Moreover, modalities are interpreted as asserting the existence of transitions which are indexed by a set of actions of which only a subset may be known. Both co-occurrence and such a sort of partial information about transitions seem to be essential for software coordination.

The rest of this chapter is organised as follows. Section 6.2 introduces modal language $\mathbb{M}$, which is used to specify interactors in section 6.3, and software connectors in section 6.4. Section 6.5 brings interactors and the coordination layer together through the notion of a *configuration*. A few examples are discussed to assess the merits of proposed approach. Finally, a few topics for future work are discussed in section 6.6.

6.2 A logic for behaviour

6.2.1 A modal language

Like many other computing artefacts, both interactors and connectors exhibit *reactive* behaviour. They evolve through reaction, either to internal events (*e.g.*, an alarm timeout) or to the accomplishment of interactions with environment (*e.g.*, the exchange of a datum in a channel end). Following a well established convention in formal modelling, we refer to all such reaction points simply as *actions*, collected on a denumerable set *Act*. Then we define modal operators which qualify the validity of logical formaluæ with respect to action occurrence, or, more generally, to action *co-occurrence*.

Having mechanisms to express *co-occurrence* becomes crucial in modelling coordination code. For example, what characterises a *synchronous channel*, the most elementary form of software glue to connect two running interactors, is precisely the fact that any interaction in its input end is simultaneous with another interaction in the output end. Note that temporal simultaneity is understood here as *atomicity*:

simultaneous actions cannot be interrupted.

The modal language introduced in the sequel is similar to the well-known Hennessy-Milner logic [HM85], but for a detail which makes it possible to express (and reason about) action *co-occurrence*. The basic idea is that a formula like $\langle a \rangle \phi$, for $a \in Act$, which in [HM85] asserts the existence of a transition indexed by a leading to a state which verifies assertion ϕ, is re-interpreted by replacing 'indexed by a' by 'indexed by a set of actions of which a is part of'. Therefore, modalities are relative to sets of actions, whose elements are represented by juxtaposition, regarded as *factors* of a (eventually larger) compound action.

In detail, modalities are indexed by either *positive* or *negative* action *factors*, denoted by K and $\sim K$, for $K \subseteq Act$, respectively. Intuitively, a *positive* (respectively, *negative*) factor refers to transitions whose labels include (respectively, exclude) all actions in it. Annotation $\sim$ may be regarded as an involution over $\mathcal{P}Act$ (therefore, $\sim\sim K = K$).

Formally $\mathbb{M}$ has the following syntax, where W is a positive or negative action factor and Ψ ranges over elementary propositions,

$$\phi \ ::= \Psi \mid \mathsf{true} \mid \mathsf{false} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \rightarrow \phi_2 \mid \langle W \rangle \phi \mid [W] \phi$$

Its semantics is given by a satisfiability relation wrt to system's states. For the non modal part this is as one would expect: for example $s \models \mathsf{true}$, $s \not\models \mathsf{false}$ and $s \models \phi_1 \wedge \phi_2 \Leftrightarrow s \models \phi_1 \ \wedge \ s \models \phi_2$. For the modal connectives, we define

$$s \models \langle W \rangle \phi \ \Leftrightarrow \ \langle \exists\, s' \ : \langle \exists\, \theta \ : \ s \xrightarrow{\theta} s' : \ W \prec \theta \rangle : \ s' \models \phi \rangle$$

$$s \models [W] \phi \ \Leftrightarrow \ \langle \forall\, s' \ : \langle \exists\, \theta \ : \ s \xrightarrow{\theta} s' : \ W \prec \theta \rangle : \ s' \models \phi \rangle$$

where

$$W \prec X \ \triangleq \begin{cases} W = K, \text{for } K \subseteq Act \ \Rightarrow \ K \subseteq X \\ W = \sim K, \text{for } K \subseteq Act \ \Rightarrow \ K \nsubseteq X \end{cases}$$

For example, if there exists a state s' such that $s \xrightarrow{abcd} s'$ and s' verifies some formula ϕ, then $s \models \langle bd \rangle \phi$. Dually, assertion $[\sim abc]\mathsf{false}$ states that all transitions whose labels do not involve, at least and simultaneously, actions in set $\{a, b, c\}$ lead to states which validate false and their occurrence is, therefore, impossible.

Modal connectives can be extended to *families* of both 'positive' or 'negative' action factors as follows:

$$s \models \langle F \rangle \phi \Leftrightarrow \langle \exists\, W \ : \ W \in F : \ \langle W \rangle \phi \rangle$$

$$s \models [F] \phi \Leftrightarrow \langle \forall\, W \ : \ W \in F : \ [W] \phi \rangle$$

where $F \subseteq (\mathcal{P}Act \cup \sim\mathcal{P}Act)$. Just as actions in an action factor are represented by juxtaposition, as in $\langle abc \rangle$, action factors in a family thereof are separated by commas, as in $\langle J, K, L \rangle$. Set complement to $\mathcal{P}Act \cup \sim\mathcal{P}Act$ is denoted by symbol $-$ as in $[-K]\mathsf{false}$ or $\langle - \rangle\mathsf{true}$, the latter abbreviating $-\emptyset$. The first assertion states that only transitions exactly labelled by factor K can occur. The second one that there exists, from the current state, at least a possible transition (of which no particular assumption is made).

Most results on Hennessy-Milner logic carry over $\mathbb{M}$. In particular, it can be shown that modal equivalence in $\mathbb{M}$ entails bisimulation equivalence for processes in CCS-like calculus extended with action co-occurrence. Although this is not the place to explore the structure of $\mathbb{M}$, the following *extension* laws are needed in the sequel:

Lemma 6.1 *For all $a, a' \in Act$, $K, K' \subseteq Act$,*

$$[a]\phi \Leftarrow [aa']\phi \tag{6.1}$$

$$\langle a \rangle\phi \Leftarrow \langle aa' \rangle\phi \tag{6.2}$$

$$[K]\phi \Leftarrow [K, K']\phi \tag{6.3}$$

$$\langle K \rangle\phi \Rightarrow \langle K, K' \rangle\phi \tag{6.4}$$

$$[K]\phi \wedge [K']\phi \Leftrightarrow [K, K']\phi \tag{6.5}$$

$$\langle K \rangle\phi \vee \langle K' \rangle\phi \Leftrightarrow \langle K, K' \rangle\phi \tag{6.6}$$

Proof. Proofs proceed by unfolding definitions. For example, (6.1) is proved as follows:

$$\begin{array}{ll}
 & s \models [a]\phi \\
\Leftrightarrow & \quad \{\text{ definition }\} \\
 & \langle \forall\, s' \;:\; \langle \exists\, \theta \;:\; s \xrightarrow{\theta} s' \;:\; \{a\} \subseteq \theta \rangle \;:\; s' \models \phi \rangle \\
\Leftarrow & \quad \{\text{ set inclusion }\} \\
 & \langle \forall\, s' \;:\; \langle \exists\, \theta \;:\; s \xrightarrow{\theta} s' \;:\; \{a, a'\} \subseteq \theta \rangle \;:\; s' \models \phi \rangle \\
\Leftrightarrow & \quad \{\text{ definition }\} \\
 & s \models [aa']\phi
\end{array}$$

□

It is also easy to see that, for K and K', both positive or both negative,

$$[K, K']\phi \;\Rightarrow\; [K \cup K']\phi \tag{6.7}$$

$$\langle K, K'\rangle\phi \;\Leftarrow\; \langle K \cup K'\rangle\phi \tag{6.8}$$

6.2.2 Typical properties

To exemplify the use of the logic and introduce some notation to be used in the sequel, let us consider a number of properties useful for the specification of both interactors and coordination schemes. Most of the latter are designed to preclude interactions in which some action factor K is absent. This leads to the following property schemes

$$\mathsf{only}\,K \;\triangleq\; [{\sim}K]\mathsf{false} \quad \text{and} \quad \mathsf{forbid}\,K \;\triangleq\; \mathsf{only}\;{\sim}K$$

Properties above entails conciseness in expression. For example, assertion $\mathsf{only}\,K \wedge \mathsf{only}\,L \wedge \mathsf{forbid}\,M$ abbreviates, by (6.5), to $\mathsf{only}\,K, L, {\sim}M$. A dual property asserts the existence of at least a transition of which a particular action pattern is a factor, *i.e.*,

$$\mathsf{perm}\,K \;\triangleq\; \langle K\rangle\mathsf{true} \tag{6.9}$$

Or, not only possible, but also mandatory,

$$\mathsf{mandatory}\,K \;\triangleq\; \langle -\rangle\mathsf{true} \wedge \mathsf{only}\,K \tag{6.10}$$

More complex patterns of behaviour are expressed by nesting modalities, as in $[K]\langle L\rangle\phi$, which expresses a sort of invariant: after every occurrence of an action with factor K, there is, at least, a transition labelled by actions in L which validates ϕ. The complement of $\langle -\rangle\mathsf{true}$ is $[-]\mathsf{false}$ which asserts no transition is possible. Notice that their duals — $\langle -\rangle\mathsf{false}$ and $[-]\mathsf{true}$ — are just abbreviations of constants false and true, respectively.

6.3 $\mathbb{M}$-interactors

6.3.1 A language for $\mathbb{M}$-interactors

As stated in the Introduction, our aim is to use a single specification notation for both *interactors*, which, in this setting, correspond to the computational entities, and *connectors*, which cater for the coordination of the former. Modal language $\mathbb{M}$

is, of course, our candidate for this double job — this section focuses on its first part.

The definition of a $\mathbb{M}$-interactor is adapted from [DH93], but for the choice of the behaviour specification language. Formally,

Definition 6.1 *An interactor signature is a triple* (S, α, Γ)*, where* S *is a set of sorts,* α *a* S*-indexed family of attribute symbols, and* Γ *a set of action symbols. An* $\mathbb{M}$*-interactor is a tuple* $(\Delta, \rho, \gamma, Ax_\Delta)$ *where* Δ *is an interactor signature,* $\rho : \mathbb{P} \longleftarrow \alpha$ *and* $\gamma : \mathbb{P} \longleftarrow \Gamma$ *are rendering relations, from attributes and actions, respectively, to some presentation medium* $\mathbb{P}$*, and* Ax_Δ *a set of axioms over* Δ *expressed in the* $\mathbb{M}$ *language.*

The set of ports provided by an interactor is defined by ρ, γ, and Γ. Ports induced by ρ are output ports used to read the value of attributes and are always available. This condition is expressed by

$$\langle \forall\ p\ :\ p \in \mathrm{ran}\ \rho :\ \langle p \rangle \mathsf{true} \rangle \tag{6.11}$$

Ports in Γ are input/output ports and their availability is governed by axioms in Ax_Δ.

Syntactically, the definition of an interactor has three main declarations: of attributes, actions and axioms. The first two define the signature. The rendering relation is given by annotations on the attributes. Actions can also be annotated to assert whether or not that they are available to the user. Fig. 6.2 shows a very simple example of an interactor modelling an application window. Two attributes are declared, indicating whether the window is visible or displays new information.

Available actions model the change of visibility and information displayed in the window. Their effect in the state of the interactor is defined by the axioms in the figure. In this example, the rendering relation is defined by the $\boxed{\text{vis}}$ annotation, which indicates that all attributes are (visually) perceivable.

Although the behavioural properties specified in this example are rather simple, in general, it is necessary to specify when actions are permitted or required to happen. This is achieved with the perm and mandatory assertions, typically stated in a guarded context. Thus,

- perm $K \rightarrow \Phi$, where Φ is a non modal proposition over the state space of the interactor, as perceived by the values of its attributes. The assertion means that *if actions containing action factor K are permitted then Φ evaluates to* true.

interactor *window*
  attributes
    $\boxed{\text{vis}}$ *visible*, *newinfo* : *bool*
  actions
    hide show update invalidate
  axioms
    [*hide*] $\neg$*visible*
    [*show*] *visible*
    [*update*] *newinfo*
    [*invalidate*] $\neg$*newinfo*
    forbid *hide show*
    forbid *update invalidate*

Figure 6.2: A window interactor

- $\Phi \rightarrow$ mandatory K, meaning *actions containing action factor K are inevitable whenever* Φ *evaluates to* true.

A useful convention establishes that permissions, but not obligations, are asserted by default. I.e., by default anything can happen, but nothing must happen. This facilitates makes adding or removing permissions and obligations incrementally when writing specifications.

6.3.2 Composing interactors

In the literature, and specifically in [CH01], interactors are composed in the 'classical' way, *i.e.*, by a specification *import* mechanism, illustrated below by means of a small example. In the literature, and specifically in [CH01], interactors are composed in the 'classical' way, *i.e.*, by a specification *import* mechanism, illustrated below by means of a small example. This will be contrasted in section 6.5 to a coordination-based solution. Consider a system that controls access to a specific space (e.g, an elevator), modelled by the interactor in Fig. 6.3. Now suppose two indicators have to be added to this model, one to announce *open* events, the other to signal *close* events. We will use instances of the window interactor from Fig. 6.2 to act as indicators. The 'classical' aggregation strategy, as in [CH01], requires that two instances of the *window* interactor be imported into one instance of *space* to build the new interactor. The rules that govern their incorporation are as follows:

- the *open* (respectively, *close*) indicator must be made visible and have its information updated whenever the system is opened (respectively, closed).

```
interactor space
  attributes
    vis state : {open, closed}
  actions
    open close
  axioms
    perm open → state = closed
    [open] state = open
    perm close → state = open
    [close] state = closed
```

Figure 6.3: The space interactor

Additionally, it should be noted that whenever a window is made visible, it might overlap (and hide) another one. The resulting interactor is presented in figure 6.4, where a new axiom expresses the coordination logic. The fact that $\mathbb{M}$ allows for action co-occurrence means that constraints on actions become simpler and more concise than their MAL counterparts, as used in [CH01]: in our example only an additional axiom is needed. Nevertheless, this solution still mixes concerns by expressing the coordination of interactors *cI* and *oI* at the same level than the internal properties of the underlying *space* interactor. How such two levels can be disentangled is the topic of the following sections.

6.4 The coordination layer

Actually, coordination entails a different perspective. As in [Arb04] this is achieved through specific *connectors* which abstract the idea of an intermediate *glue code* to handle interaction. Connectors have *ports*, thought of as *interface points* through which messages flow. Each port has an *interaction polarity* (either *input* or *output*), but, in general, connectors are blind with respect to the data values flowing through them. The set of elementary interactions of a connector $\mathbb{C}$ forms its *sort*, denoted by $\mathsf{sort}.[\![\mathbb{C}]\!]$. By default the sort of $\mathbb{C}$ is the set of its ports, but often such is not the case. For example, a synchronous channel with ports a and a' has a unique possible interaction: the simultaneous activation of both a and a', represented by aa'.

Connectors are specified at two levels: the *data* level, which records the flow of data, and the *behavioural* one which prescribes all the activation patterns for ports. Formally, let $\mathbb{C}$ be a connector with m input and n output ports. Assume $\mathbb{D}$ as a generic type of data values and $\mathbb{P}$ as a set of (unique) *port identifiers*. Then,

interactor *spaceSign*

  aggregates

    window **via** *oI*

    window **via** *cI*

  attributes

    $\boxed{\text{vis}}\ state : \{open, closed\}$

  actions

    $\boxed{\text{vis}}\ open\ close$

  axioms

    $\mathsf{perm}\ open \rightarrow state = closed$

    $[open]\ state' = open$

    $\mathsf{perm}\ close \rightarrow state = open$

    $[close]\ state' = closed$

    $\mathsf{only}\ open\ oI.update\ oI.show\ \vee\ \mathsf{only}\ close\ cI.update\ cI.show$

Figure 6.4: A classical solution

Definition 6.2 *The specification of a connector* $\mathbb{C}$ *is given by a relation* $\mathsf{data}.[\![\mathbb{C}]\!] : \mathbb{D}^n \longleftarrow \mathbb{D}^m$*, which relates data present at its m input ports with data at its n output ports, and an* $\mathbb{M}$ *assertion,* $\mathsf{port}.[\![\mathbb{C}]\!]$*, over its sort,* $\mathsf{sort}.[\![\mathbb{C}]\!]$*, which specifies the relevant properties of its port activation pattern.*

6.4.1 Elementary connectors.

The most basic connector is the *synchronous channel* which exhibits two ports, a and a', of opposite polarity. This connector forces input and output to become mutually blocking. Formally, $\mathsf{data}.[\![\ a \longrightarrow a'\ ]\!] = \mathsf{Id}_{\mathbb{D}}$, i.e., the identity relation in $\mathbb{D}$, and

$$\mathsf{sort}.[\![\ a \longrightarrow a'\ ]\!] = \{aa'\} \quad \mathsf{port}.[\![\ a \longrightarrow a'\ ]\!] = \mathsf{only}\ aa'$$

Its static semantics is simply the identity relation on data domain $\mathbb{D}$ and its behaviour is captured by the simultaneous activation of its two ports.

Any coreflexive relation provides channels which can loose information, thus modelling unreliable communications. Therefore, we define, an *unreliable channel* as $\mathsf{data}.[\![\ a \xrightarrow{\diamond} a'\ ]\!] \subseteq \mathsf{Id}_{\mathbb{D}}$ and

$$\mathsf{sort}.[\![\ a \xrightarrow{\diamond} a'\ ]\!] = \{a, aa'\} \quad \mathsf{port}.[\![\ a \xrightarrow{\diamond} a'\ ]\!] = \mathsf{only}\ a$$

The behaviour expression states that all valid transitions involve input port a, although not necessarily a'. This corresponds either to a successful communication, represented by the simultaneous activation of both ports, or to a failure, represented by the single activation of the input port.

As an example of a connector which is not stateless consider *fifo*$_1$, a channel with a buffer of a single position. Formally, $\mathsf{data}.[\![\ a \xrightarrow{\ \Box\ } a'\ ]\!] \ = \ \mathsf{Id}_{\mathbb{D}}$ and

$$\mathsf{sort}.[\![\ a \xrightarrow{\ \Box\ } a'\ ]\!] \ = \ \{a, a'\} \quad \mathsf{port}.[\![\ a \xrightarrow{\ \Box\ } a'\ ]\!] \ = \ [a]\mathsf{only}\, a', {\sim}a$$

Notice that its port specification equivales to $[a](\mathsf{only}\, a' \wedge \mathsf{forbid}\, a)$, formalising the intuition of a strict alternation between the activation of ports a and a'.

If channels forward information, *drains* absorb it. However they play a fundamental role in controlling the flow of data along the coordination code. A *drain* has two input, but no output, ports. Therefore, it looses any data item crossing its boundaries. A drain is *synchronous* if both write operations are requested to succeed at the same time (which implies that each write attempt remains pending until another write occurs in the other end-point). It is *asynchronous* if, on the other hand, write operations in the two ports do not coincide. The data part coincides in both connectors: $\mathbb{D} \times \mathbb{D}$. Then

$$\mathsf{sort}.[\![\ a \vdash\!\!\!\longrightarrow\!\!\!\dashv a'\ ]\!] \ = \ \{aa'\} \quad \mathsf{port}.[\![\ a \vdash\!\!\!\longrightarrow\!\!\!\dashv a'\ ]\!] = \mathsf{only}\, aa'$$

$$\mathsf{sort}.[\![\ a \overset{\triangledown}{\vdash\!\!\!\longrightarrow\!\!\!\dashv} a'\ ]\!] \ = \ \{a, a'\} \quad \mathsf{port}.[\![\ a \overset{\triangledown}{\vdash\!\!\!\longrightarrow\!\!\!\dashv} a'\ ]\!] = \mathsf{only}\, a, a' \ \wedge \ \mathsf{forbid}\, aa'$$

6.4.2 New connectors from old

Connectors can be combined in three different ways: by placing them side-by-side, by sharing ports or introducing feedback wires to connect output to input ports. In the sequel, note that behaviour annotations in the specification of connectors can always be presented in a *disjunctive* form

$$\mathsf{port}.[\![\mathbb{C}]\!] \ = \ \phi_1 \vee \phi_2 \vee \cdots \vee \phi_n \tag{6.12}$$

where each ϕ_i is a conjunction of

$$\underbrace{[K] \cdots [K]}_{n} \ \mathsf{only}\, F$$

Also let $t_{|a}$ and $t_{\#a}$, for $t \in \mathbb{D}^n$ and $a \in \mathcal{P}$, represent, respectively, a tuple of data values t from which the data corresponding to port a has been deleted, and the tuple component corresponding to such data. Then,

Join. This combinator places its arguments side-by-side, with no direct interaction between them. Then,

$$\begin{aligned}
\mathsf{data}.[\![\mathbb{C}_1 \boxplus \mathbb{C}_2]\!] &= \mathsf{data}.[\![\mathbb{C}_1]\!] \times \mathsf{data}.[\![\mathbb{C}_2]\!] \\
\mathsf{sort}.[\![\mathbb{C}_1 \boxplus \mathbb{C}_2]\!] &= \mathsf{sort}.[\![\mathbb{C}_1]\!] \cup \mathsf{sort}.[\![\mathbb{C}_2]\!] \\
\mathsf{port}.[\![\mathbb{C}_1 \boxplus \mathbb{C}_2]\!] &= \mathsf{port}.[\![\mathbb{C}_1]\!] \vee \mathsf{port}.[\![\mathbb{C}_2]\!]
\end{aligned}$$

The relevance of sorts becomes now clear. Take, for example, the aggregation of two synchronous channels Their joint behaviour is

$$\mathsf{port}.[\![(\, a \longrightarrow a' \boxplus c \longrightarrow c' \,)]\!] \;=\; \mathsf{only}\, aa' \vee \mathsf{only}\, cc'$$

A transition labelled by, say, $aa'c$ does not violate the behaviour prescribed above, but it is made invalid by the sort specification, which is $\{aa', cc'\}$.

Share. The effect of *share* is to plug ports with identical polarity. The aggregation of *output* ports is done by a *right share* ($\mathbb{C}\ {}^{i}_{j}\!> z$), where $\mathbb{C}$ is a connector, i and j are ports and z is a fresh name used to identify the new port. Port z receives asynchronously messages sent by either i or j. When input from both ports is received at same time the combinator chooses one of them in a non-deterministic way. Let $\mathsf{data}.[\![\mathbb{C}]\!] : \mathbb{D}^n \longleftarrow \mathbb{D}^m$. Then, the data flow relation $\mathsf{data}.[\![\mathbb{C}\ {}^{i}_{j}\!> z]\!] : \mathbb{D}^{n-1} \longleftarrow \mathbb{D}^m$ for this operator is given by

$$r\,(\mathsf{data}.[\![\mathbb{C}\ {}^{i}_{j}\!> z]\!])\, t \;\Leftrightarrow\; t'\,(\mathsf{data}.[\![\mathbb{C}]\!])\, t \;\wedge\; r_{|z} = t'_{|i,j} \;\wedge\; (r_{\#z} = t'_{\#i} \;\vee\; r_{\#z} = t'_{\#j})$$

At the behavioural level, its effect is that of a renaming applied to the $\mathbb{M}$-formula capturing the behavioural patterns of $\mathbb{C}$, *i.e.*,

$$\mathsf{port}.[\![(\mathbb{C}\ {}^{i}_{j}\!> z)]\!] \;=\; \{z \leftarrow i, z \leftarrow j\}\ \mathsf{port}.[\![\mathbb{C}]\!]$$

over

$$\mathsf{sort}.[\![(\mathbb{C}\ {}^{i}_{j}\!> z)]\!] \;=\; \{z \leftarrow i, z \leftarrow j\}\ \mathsf{sort}.[\![\mathbb{C}]\!]$$

Figure 6.5 represents a *merger* formed by sharing the output ports of a synchronous channel and a 1-place buffer.

On the other hand, aggregation of *input* ports is achieved by a *left share* mechanism ($z <^{i}_{j} \mathbb{C}$). This behaves like a *broadcaster* sending synchronously messages from z to both i and j. This case is slightly more complex: before renaming, all computations of $\mathbb{C}$ in which ports i and j are activated independently of each other must be synchronised. Therefore, we take all disjuncts in $\mathsf{port}.[\![\mathbb{C}]\!]$ in which ports i

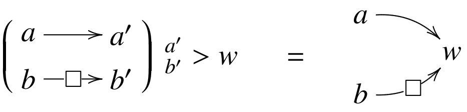


Figure 6.5: A *merger*: $\mathsf{only}\, aw \vee [b]\mathsf{only}\, w, {\sim}b$.

and j are involved, form their conjunction to force co-occurrence, and apply renaming. Formally, let ϕ_θ be a disjunct in $\mathsf{port}.[\![\mathbb{C}]\!]$ (recall (6.12)) involving only ports in θ. Define $\phi_i = \langle \bigvee\ \phi_\theta \in \mathsf{port}.[\![\mathbb{C}]\!]\ :\ i \in \theta\ :\ \phi_\theta \rangle$ and, similarly, ϕ_j. Therefore, for $\sigma = \{z \leftarrow i, z \leftarrow j\}$,

$$\mathsf{port}.[\![(z <^i_j \mathbb{C})]\!] \;=\; \sigma(\phi_i \wedge \phi_j) \;\vee\; \langle \bigvee\ \phi_{\theta'} \in \mathsf{port}.[\![\mathbb{C}]\!]\ :\ i \notin \theta' \wedge j \notin \theta'\ :\ \phi_{\theta'} \rangle$$

and, again,

$$\mathsf{sort}.[\![(z <^i_j \mathbb{C})]\!] \;=\; \{z \leftarrow i, z \leftarrow j\}\ \mathsf{sort}.[\![\mathbb{C}]\!]$$

On the other hand, relation $\mathsf{data}.[\![z <^i_j \mathbb{C}]\!] : \mathbb{D}^n \longleftarrow \mathbb{D}^{m-1}$ is given by

$$t'\,(\mathsf{data}.[\![z <^i_j \mathbb{C}]\!])\,r \;\leftrightarrow\; t'\,(\mathsf{data}.[\![\mathbb{C}]\!])\,t \;\wedge\; r_{|z} = t_{|i,j} \;\wedge\; r_{\#z} = t_{\#i} = t_{\#j}$$

As an example let us calculate the sharing of input ports a and b in a connector composed by three, otherwise non interfering, synchronous channels,

$$\begin{array}{ll}
& \mathsf{port}.[\![z <^a_b (\ a \longrightarrow a'\ \boxplus\ b \longrightarrow b'\ \boxplus\ c \longrightarrow c'\ )]\!] \\
\Leftrightarrow & \qquad \{\ \text{definition}\ \} \\
& \{z \leftarrow a, z \leftarrow b\}(\mathsf{only}\, aa' \wedge \mathsf{only}\, bb') \vee \mathsf{only}\, cc' \\
\Leftrightarrow & \qquad \{\ \text{renaming and (6.5)}\ \} \\
& \mathsf{only}\, za', zb' \vee \mathsf{only}\, cc' \\
\Leftrightarrow & \qquad \{\ (6.7)\ \} \\
& \mathsf{only}\, za'b' \vee \mathsf{only}\, cc'
\end{array}$$

which asserts that input on z co-occurs with output at both a' and b'. Replacing $b \longrightarrow b'$ by a one-place buffer leads to the connector depicted in Fig. 6.6 which

$$z <^a_b \begin{pmatrix} a \longrightarrow a' \\ b \xrightarrow{\square} b' \\ c \longrightarrow c' \end{pmatrix} \quad = \quad \begin{tikzcd} & a' \\ z \arrow[ur] \arrow[dr, "\square"'] & \\ & b' \\ c \arrow[r] & c' \end{tikzcd}$$

Figure 6.6: A *broadcaster* and a detached channel.

is calculated as follows

$$\begin{array}{ll} & \mathsf{port}.[\![z <^a_b (\, a \longrightarrow a' \boxplus b \xrightarrow{\square} b' \boxplus c \longrightarrow c' \,)]\!] \\ \equiv & \qquad \{ \text{ definition } \} \\ & \{z \leftarrow a, z \leftarrow b\}(\mathsf{only}\, aa' \wedge [b]\mathsf{only}\, b', {\sim}b) \vee \mathsf{only}\, cc' \\ \equiv & \qquad \{ \text{ renaming } \} \\ & (\mathsf{only}\, za' \wedge [z]\mathsf{only}\, b', {\sim}b) \vee \mathsf{only}\, cc' \end{array}$$

Hook. This combinator encodes a *feedback* mechanism, drawing a direct connection between an output and an input port. This has a double consequence: the connected ports must be activated simultaneously and become externally non observable.

Formally, such conditions are expressed in $\mathsf{port}.[\![\mathbb{C} \Lsh^j_i]\!]$ applying to $\mathsf{port}.[\![\mathbb{C}]\!]$ an operation $\Lsh^j_i$ on assertions defined as follows[1]. First notice that every ϕ_k in the disjunctive form of $\mathsf{port}.[\![\mathbb{C}]\!]$ in (6.12), is a conjunction of modal assertions

$$\mu_l = \underbrace{[K] \cdots [K]}_{n} \mathsf{only}\, F$$

for $n \geq 0$. Thus, let $\Phi = \mathsf{port}.[\![\mathbb{C}]\!]$ and compute $\Phi \Lsh^j_i$ as follows:

1. Remove from Φ all assertions μ_i and μ_j which involve at least an occurrence of action i or j, respectively. Let Ψ be the remaining formula, *i.e.*, the original Φ where all removed μ are replaced by the relevant identity (either true or false).

[1] The definition of *hook* given below is not entirely general but can handle most cases of interest in coordination problems and all examples given here. Notice, in particular, all cases symmetric in ϕ and ψ are omitted to enhance readability.

2. For all μ involving simultaneously actions i and j, compute $\gamma = \{\emptyset \leftarrow i, \emptyset \leftarrow j\}\,\mu$.

3. For all pairs μ_i and μ_j, involving i and j, respectively, compute $\gamma = \overline{\mu_i \mu_j}$ by

$$\begin{cases} \mu_i = \mathsf{only}\, iK \wedge \mu_j = \mathsf{only}\, jL \;\rightsquigarrow\; \mathsf{only}\, K, L \\ \mu_i = \mathsf{only}\, iK \wedge \mu_j = \underbrace{[j]\cdots[j]}_{n}\,\mathsf{only}\, L, {\sim}j \;\rightsquigarrow \\ \qquad \underbrace{[K]\cdots[K]}_{n}\,\mathsf{only}\, L, {\sim}K \\ \mu_i = \underbrace{[K]\cdots[K]}_{m}\,\mathsf{only}\, i \wedge \mu_j = \mathsf{only}\, jL \;\rightsquigarrow\; \underbrace{[K]\cdots[K]}_{m}\,\mathsf{only}\, L \\ \mu_i = \underbrace{[K]\cdots[K]}_{m}\,\mathsf{only}\, i \wedge \mu_j = \underbrace{[j]\cdots[j]}_{n}\,\mathsf{only}\, L, {\sim}j \;\rightsquigarrow \\ \qquad \underbrace{[K]\cdots[K]}_{m+n}\,\mathsf{only}\, L, {\sim}K \end{cases}$$

4. Let Γ be the conjunction of all formulas γ computed in steps 2, 3 and 4. Then define $\Phi \Lsh^{j}_{i} = \Gamma \wedge \Psi$.

Note that γ assertions obtained in step 2. reduce to either true or false. It is instructive to compute the result of hooking of a synchronous channel and of a 1-place buffer. In the first case we get

$$(\mathsf{only}\, aa') \Lsh^{a}_{a'} \;=\; \mathsf{only}\, \emptyset \;=\; \mathsf{true}$$

In the second,

$$[a](\mathsf{only}\, a', {\sim}a) \Lsh^{a}_{a'} \;=\; [\emptyset](\mathsf{true} \wedge \mathsf{false}) \;=\; \mathsf{false}$$

as one may have expected given the buffer strict alternation activation discipline.

Clearly the sort of $\mathbb{C} \Lsh^{j}_{i}$ is obtained from that of $\mathbb{C}$ by consistently removing from each elementary interaction port identifiers i and j. On the other hand, the effect of *hook* on data, assuming $\mathsf{data}.[\![\mathbb{C}]\!] : \mathbb{D}^n \longleftarrow \mathbb{D}^m$, is modelled by relation

$$\mathsf{data}.[\![\mathbb{C} \Lsh^{j}_{i}]\!] : \mathbb{D}^{n-1} \longleftarrow \mathbb{D}^{m-1}$$

specified by

$$t'_{|j}\,(\mathsf{data}.[\![\mathbb{C} \Lsh^{j}_{i}]\!])\, t_{|i} \quad \text{iff} \quad t'\,(\mathsf{data}.[\![\mathbb{C}]\!])\, t \;\wedge\; t'_{\#j} = t_{\#i}$$

Simple calculations show that the synchronous channel is the identity for *hook* or that plugging two 1-place buffers sequentially produces a 2-place buffer.

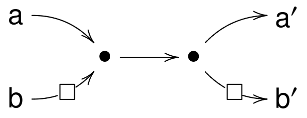


Figure 6.7: An example of *hook* usage.

As a less obvious example consider hooking a *merger* $\mathbb{M}$ and a *broadcaster* $\mathbb{B}$ (as represented in Fig. 6.5 and the upper part of Fig. 6.6). The resulting connector is depicted in Fig. 6.7 and its behaviour computed by

$$
\begin{array}{ll}
 & \mathsf{port}.\llbracket(\mathbb{M} \boxplus \mathbb{B}) \Lsh^{w}_{z}\rrbracket \\
\equiv & \qquad \{\ \text{definition}\ \} \\
 & ((\mathsf{only}\, aw \vee [b]\mathsf{only}\, w, {\sim}b) \ \vee\ (\mathsf{only}\, za' \wedge [z]\mathsf{only}\, b', {\sim}z)) \Lsh^{w}_{z} \\
\equiv & \qquad \{\ \textit{hook}\ \text{definition}\ \} \\
 & \mathsf{only}\, aa' \wedge [a]\mathsf{only}\, b', {\sim}a \wedge [b]\mathsf{only}\, a', {\sim}b \wedge [b][b]\mathsf{only}\, b', {\sim}b
\end{array}
$$

6.5 Configurations of $\mathbb{M}$-interactors

Having introduced $\mathbb{M}$-interactors and the coordination layer on top of the same modal language, we may now complete the whole picture. The key notion is that of a *configuration*, *i.e.*, a collection of *interactors* interconnected through a *connector* built from elementary connectors, combined trough the combinators defined above. Formally,

Definition 3 *A configuration is a tuple*

$$\langle I, \mathbb{C}, \sigma \rangle \tag{6.13}$$

where $I = \{I_i |\ i \in n\}$ *is a collection of interactors,* $\mathbb{C}$ *is a connector and* σ *a mapping of ports in* I *to ports in* $\mathbb{C}$*. The behaviour of a configuration is given by the conjunction of the modal theories for each* $I_n \in I$*, as specified by their axioms, and the port specification* $\mathsf{port}.\llbracket\mathbb{C}\rrbracket$ *of connector* $\mathbb{C}$*, after renaming by* σ*.*

To illustrate the envisaged approach, consider again the example discussed in section 6.3. A coordination-based solution, depicted in Fig. 6.8, replaces the hierarchical import of *window* into *spaceSign* interactor, by a configuration in which

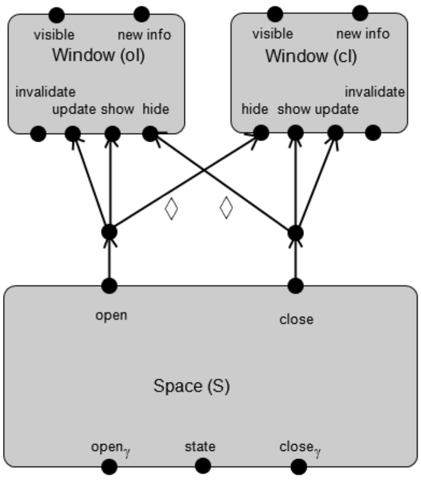


Figure 6.8: A coordination-based solution.

the two instances of the former and one instance of the original *space* interactor are connected by

$$\mathbb{BC} \triangleq \mathbb{B} \boxplus \mathbb{B}$$

a connector which joins together two broadcasters $\mathbb{B}$. Each $\mathbb{B}$ is formed by two synchronous channels and a lossy channel, sharing their input ports, *i.e.*

$$\mathbb{B} \triangleq z <^w_c (w <^a_b (a \longrightarrow a' \boxplus b \longrightarrow b') \boxplus c \xrightarrow{\diamond} c')$$

An easy calculation yields $\mathsf{port}.[\![\mathbb{B}]\!] = \mathsf{only}\, za', zb', z$, which, by (6.7), equivales to $\mathsf{only}\, za'b'$. In a configuration in which, through a renaming σ, port z is linked to $S.open$, a' to $oI.update$, b' to $oI.show$ and c' to $cI.hide$, one may prove (*i.e.*, discover, rather than assert) a number of desirable properties of the configuration. For example, from axiom $\mathsf{perm}\, S.open$, a default axiom of interactor *space* in section 6.3, and $\sigma\, \mathsf{only}\, za'b'$, one concludes that

$$\mathsf{perm}\, S.open\ oI.update\ oI.show$$

i.e., there are transitions in which all the three ports are activated at the same time. But, because the connector does not allow actions not including the simultaneous activation of such three ports, the joint behaviour of the configuration asserts not only possibility but also necessity of this transition, *i.e.*,

$$\mathsf{perm}\, S.open\ oI.update\ oI.show \ \wedge\ \mathsf{only}\, S.open\ oI.update\ oI.show$$

This is stronger than the corresponding axiom added to interactor *spaceSign* in Fig. 6.4, although it can be deduced from the modal theory of this interactor (which,

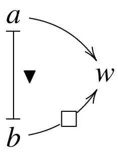


Figure 6.9: An *alternate merger*.

of course, includes $\mathsf{perm}\, open$). Note we are focussing only on one of the two $\mathbb{B}$ connectors in $\mathbb{BC}$, thus this conclusion does not interfere with a similar possibility for the other connection of interactor instances S and cI (recall the behavioural effect of $\boxplus$ is disjunction).

On the other hand, one also has

$$\mathsf{perm}\, S.open\; cI.hide$$

because action zc' is in $\mathsf{sort}.[\![\mathbb{B}]\!]$, but, now only as a possibility, because an unreliable channel was used to connect these ports. From this property and

$$\mathsf{only}\, S.open\; oI.update\; oI.show$$

above, we can easily conclude that $cI.hide$ cannot occur independently of $S.open$, $oI.update$ and $oI.show$. Again, this is stronger than the interactor model in Fig. 6.4, where the hide action was left unrestricted.

As a final example, consider an interactor which has to receive the location coordinates supplied by two different input devices but in strict alternation. The connector to plug these three interactors is the *alternate merger* depicted in Fig. 6.9, formally, defined as

$$b <^{d'}_{f} a <^{d}_{c} (\, c \longrightarrow c' \;\boxplus\; d \xrightarrow{\blacktriangledown} d' \;\boxplus\; f \,\text{—}\square\!\!\rightarrow f')^{c'}_{f'} > w$$

Its behavioural pattern is

$$\mathsf{port}.[\![\mathbb{AM}]\!] \;=\; \mathsf{only}\, awb \,\wedge\, [b]\mathsf{only}\, w, {\sim}b$$

Clearly, each activation of port a is synchronous with b and w. But then data received in b (say, the coordinates of the one of the devices) is stored in the buffer. Next action is necessarily w, whose completion empties the buffer.

6.6 Concluding

It was the intention of this case-study to set the foundations for an approach to modelling interactive systems entailing a true separation of concerns between modelling of individual components (interactors) and their architectural organisation. For this a new modal logic (the $\mathbb{M}$ language) was introduced, which is similar to the Hennessy-Milner logic [HM85] but for the fact that its modal connectives are indexed by sets of actions (*actions factors*). These action factors are interpreted over the compound actions (themselves represented by sets) that label transitions using set inclusion. This makes it possible to express properties over co-occurring actions in the logic.

Although the main drive behind the development of $\mathbb{M}$ was the need for a modal logic expressive enough to define the coordination layer, the language was also used to specify interactors, thus providing a single language for expressing the behaviour of both interactors and connectors that bind them.

The approach presents two major benefits over [FP90] or [CH01]. First of all, it promotes a clear separation of concerns between the specification of the individual interactors and the specification of how they interact with each other. Furthermore, it frees us from the rigid structure imposed by hierarchical organisation.

At this point, it is worthwhile pointing out that when composing interactors into different configurations, the resulting behaviour becomes an emergent feature of the model. Hence, we discover, rather than assert, what the system will be like. This is particularly relevant in a context were one is interested in exploring the impact of different design decisions at the architectural level. Recent related work on the use of (some type of) logic to specify component behaviour include [BM07] and [JOT06], the latter with an emphasis on property verification.

A number of lines of research have been opened by the current endeavour. A main one concerns temporal extension. Actually, language $\mathbb{M}$ seems expressive enough to express connector's behaviour, but not so when facing more elaborate interactor's specifications. A typical case relates to expressing *obligation* requirements. This entails the need for studying how $\mathbb{M}$ can be extended in a way similar to D. Kozen's μ-calculus [Koz83] in order to address these temporal issues.

Chapter 7

A Functional Library for Prototyping Software Connectors

Summary

This chapter introduces a set of basic programming primitives which implement some of the main concepts and models described in this thesis, for prototyping purposes. Components' interfaces and connectors are encoded using the functional language HASKELL. *Component interfaces are built from generic ports allowing anonymous communication among components. As discussed in the previous chapters, a connector is the device responsible for orchestrating such sort of component interaction. Although not expected to be a proof-of-concept for the models introduced in the thesis, this library allows the software architect to 'play' with connectors and configurations when designing a new system.*

7.1 Functional components and their interfaces

This chapter gives a glimpse of a functional library developed with the purpose of animating software connectors and their composition.

The language used to implement the library HASKELL [Bir98]. HASKELL is a powerfull language, endowed with several specific extensions and libraries. In particular, we resorted to CONCURRENT HASKELL, a expressive extension of HASKELL, with explicit concurrency [JGF96].

Components

The library Interface allows us create functional components, *i.e.*, it provides a way to wrap functional programs, giving to it a public interface specifying ports which will be visible to the environment.

The data type Port is defined as follows,

```
data Port a
 = Port (MVar a)
        (MVar ())
```

In order to allow persistence and generality of data values, ports are implemented as mutable variables, *i.e.*, MVars, which are primitives of the Concurrent Haskell [JGF96]. In fact, a MVar is a reference to a mutable location that either can contain a value of type *a*, or can be empty.

The creation of a port is done by invoking the operation createPort. Such an operation is defined by,

```
createPort :: IO (Port a)
createPort
 = newEmptyMVar >>= \ datum ->
   newMVar ()   >>= \ ack ->
   return (Port datum ack)
```

Associated to a port there are a number of operations to read and write from/to it. Such operations are: readPort, defined by,

```
readPort :: Port a -> IO a
readPort (Port datum ack)
 = takeMVar datum >>= \ val ->
   putMVar ack () >>
   return val
```

and writePort, whose definition is given by

```
writePort :: Port a -> a -> IO ()
writePort (Port datum ack) val
 = takeMVar ack      >>
   putMVar datum val >>
   return ()
```

Example 7.1 *A very simple example of the creation and the usage of a port is given in the following,*

```
prt1 = unsafePerformIO $ createPort
send pt msg = do
                writePort pt msg
              receive
receive = do
                x <- readPort p1
             print (x)
main = do
       forkIO (send prt1 123)
```

In this example a port $prt1$ *is created and the function send writes a numeric value which is read in the function receive. Notice this is just a very simple producer/consumer example using basically one port.*

7.2 Connectors

Let us now focus on the creation of mechanisms allowing components to mutually cooperate and change information in order to achieve some common goal. In order to do that, we have a set of primitives which allow us to create basic simple connectors (or channels, in this case, enforcing a point–to–point anonymous communication).

Connectors also have ports, *i.e.*, regulated openings through which they exchange units of information.

The library Connectors provides the abstract data type *Port* which allow us to create ports uniquely identified. The data type port is given by,

```
data Port a = Prt {idp :: PortId, hole :: MVar (Stream a)}
type PortId = String
```

where `idp` is a port identifier and `hole` is a mutable variable through which the messages will flow.

Basic Channels

Let us call the most basic connectors (without any way of composition) by basic channels. The *synchronous* and the *asynchronous* channels are the most well-known examples of connectors of this kind. Both of them are connectors with a single input port and a single output port. The difference relies on the absence or presence of buffering capabilities.

A *channel* shares, in fact, the same data type, *i.e.*, a channel is a structure with two ports and the data flowing through such a structure must be the same at both input and output ports. The difference between a synchronous and an asynchronous channel is in the obligation to synchronize or not. The library MyQsem is responsible to ensure the synchronization constraints in case of a synchronous channel. Such a library provides a semaphore which regulates the order of writing and reading operation to/from ports, ensuring then the correct flow of messages.

The library provides the following basic channels:
Sync: *Sync* is a *synchronous channel*, it has an input and an output port and the I/O operations occur simultaneously. This connector has no buffering capabilities. It is defined as follows,

```
newSync :: PortId -> PortId -> IO (Channel i o)
newSync idw idr = do
   buffer <- newEmptyMVar
   read1 <- newMVar buffer
   write1 <- newMVar buffer
   canI' <- CC.newQSem 0
   wsyncL <- newMVar []
   rsyncL <- newMVar []
   rep <- newMVar 1
   oo <- newMVar True
   return Channel { writeC = [Prt {idp=idw, hole=write1}],
                    readC = [Prt {idp=idr, hole=read1}],
                    fC = id, canI = canI', writeSyncList = wsyncL,
                    readSyncList = rsyncL, writeRep = rep, readRep = rep,
                    writeSync = rep, readSync = rep,
                    onlyOne = oo
                  }
```

Considering that the connectors obey the same principle of construction the synchronous connector is enough to demonstrate the construction of a connector. The particular feature of each channel will be regulated by the semaphore implemented in the function *canI* which will regulate the synchrony of data flow when necessary.

Async: The *Async* type represents an *asynchronous connector*. It has an input and an output port and unlimited buffering capabilities. The I/O operations occur at different time.

Lossy: The *lossy connector* is able to discard some messages in a controlled way. Actually the lossy connector behaves exactly as a synchronous channel when both operations, read and write, succeed at same time, on the other hand, when the operations do not coincide the data value pending on the connector is lost.

Asyncdrain: Represents an *asynchronous drain connector*. It is a connector with two inputs and no outputs. The input operations must occur at different time. This connectors loses every message which comes into it.

Syncdrain: Represents a *synchronous drain connector*. It is a connector with two inputs and no outputs and the input operations must occurs at a same time.

Concentrator: The *concentrator connector* is a ternary connector with two inputs and a single output. In this connector the pair of inputs and the output operations must occur simultaneously.

Merger: The *merger connector* is very similar to the concentrator connector, but in this case only one input and the corresponding output may occur at same time.

SyncBroadcaster: The *synchronous broadcaster* is a connector with a single input operation and two output operations. The input operation and the output operations must occur simultaneously.

New connectors are created using the function `createChannel` defined as follows,

```
createChannel :: NewChannel i o -> IO (Channel i o)
createChannel (Sync w r) = newSync w r
createChannel (Async w r) = newAsync w r
createChannel (Lossy w r f) = newLossy w r f
createChannel (Merger w1 w2 r) = newMerger w1 w2 r
createChannel (Concentrator w1 w2 r) = newConcentrator w1 w2 r
createChannel (SyncBroadcaster w r1 r2) = newSBroadcaster w r1 r2
createChannel (AsyncDrain w1 w2) = newADrain w1 w2
createChannel (SyncDrain w1 w2) = newSDrain w1 w2
```

The data type NewChannel is defined by,

```
data NewChannel i o =
                  Sync PortId PortId
                | Async PortId PortId
                | Lossy PortId PortId (i -> o)
                | Merger PortId PortId PortId
                | Concentrator PortId PortId PortId
                | SyncBroadcaster PortId PortId PortId
                | AsyncDrain PortId PortId
                | SyncDrain PortId PortId
```

So, in order to construct a new channel we must proceed according to following command

```
chan <- createChannel (Sync "p1" "2")
```

, *i.e.*, we pass as arguments the type of the desired connector, $Sync$ in this example and the names of its ports, $p1$ and $p2$.

Basically a channel will be composed by two or three ports and, according to the polarity of data flow (input or output), the channel will be formed. For instance, a binary *sync* channel is composed by an input port and an output port, or a *syncdrain* channel, also a binary channel but, composed by two input ports. The *merger*, *concentrator* and *SyncConcentrator* are examples of ternary channels.

The input and output operations on connectors are provided by the input operation *post* and the output operation *read*.

The *post* operation is defined by,

```
post :: (Eq i, Show i) => Channel i o -> i -> PortId -> IO ()
post (Channel w r _ canI acts _ rep _ max _ oo) val idr =
block $ do
   CC.waitQSem canI  -- espaço?
   rep' <- readMVar rep
   max' <- readMVar max
   if max' /= 1 then do
      rep'' <- verActW' idr acts [] max' val rep' o
      writeAction w idr val rep''
   else do
      writeAction w idr val rep'
   where
mapList (f:fs) (x:xs) = (f x) : (mapList fs xs)
mapList _ _ = []
mapIO f [] _ = return ()
mapIO f (x:xs) (p:ps) = do
     x' <- x
     f' f x' p >> mapIO f xs ps
    where f' _ [] _ = return ()
       f' f (h:t) p = f h p >> f' f t p
```

The *read* operation is defined by,

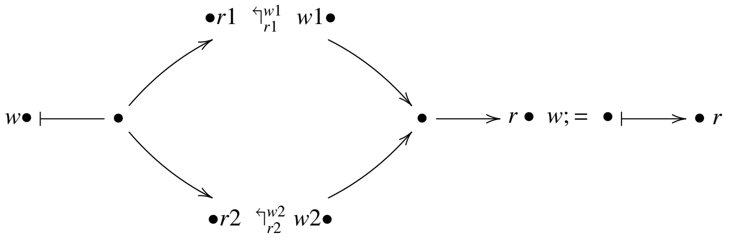
```
_read :: (Eq i, Show o) => Channel i o -> PortId -> IO [o]
_read ch@(Channel writeC readC f canI _ acts _ rep _ max _) idr
  =
  block $ do
   CC.signalQSem canI
   rep' <- readMVar rep
   max' <- readMVar max
   if max' /= 1
      then do
        verActR' idr acts [] rep'
        x <- readAction readC idr f rep' []
        return x
      else do
        x <- readAction readC idr f rep'
        return x
```

7.3 Combinators

The essence of coordination models is the capability of to construct more elaborated connectors, from most basic channels. In our case, complex connectors may be created aggregating or plugging ports with opposite polarity using the primitive hook. In the follow we have the hook operation described,

```
hook :: (Eq o, Show o, Show k)
    =>
    Channel o o -> Channel o k -> [(String, String)] -> IO (Channel o k)
    hook ch1@(Channel w1 r1 )
         ch2@(Channel w2 r2 ) l
    =
     if l == []
        then
            error "no ports available..."
        else
        let readPrts = map fst l
            writePrts = map snd l
        in if not (isAllIn readPrts (map idp r1)) ||
              not (isAllIn writePrts (map idp w2))
            then
              error "incompatible ports..."
            else do
                forkIO $ readWrite readPrts writePrts
                let w1' = getNotUsedPorts writePrts $ w1++w2
                let r2' = getNotUsedPorts readPrts $ r1++r2
                return $ Channel w1' r2' (a2.a1) b1 c1 d2 e1 f2 g1 h2 i1
    where
        readWrite readPrts writePrts = do
        runInBoundThread $ do
                    xt <- barrier $ mapM__ (_read ch1) readPrts
                    let xt2 = (map_ (map (post_ ch2) writePrts)) xt
                    ...
```

The *hook* combinator receives the ports to be plugged and returns a new connector hidding the ports which were hooked. Let us consider an example of the usage of the hook constructing a synchronous connector by plugging a synchronous broadcaster and a concentrator.



Consider the following function *testS BroadConce*,

```
testSBroadConce = do
                    ch1 <- createChannel (SyncBroadcaster "w" "r1" "r2")
                    ch2 <- createChannel (Concentrator "w1" "w2" "r")
                    ch <- hook ch1 ch2 [("r1", "w1"), ("r2", "w2")]
                    forkIO $ post ch 'a' "w"
                    forkIO $ _read ch "r"
```

first the broadcaster connector *ch*1 is created with the input port *w* and two output ports *r*1 and *r*2. After this, the concentrator connector *ch*2 is created with the input ports *w*1 and *w*2 and the output port *r*. The hook combinator plugs the ports *r*1, *w*1 and *r*2, *w*2, creating the connector *ch*. This operation hides the ports which were hooked and the visible resulting ports are the input port *w* and the output ports *r*.

Example 7.2 *As an example, let us consider the elementary bank system performing a backup operation (see Fig. 5.8 in chapter 5). We have three components, DBRep, ATM and Bank, and a connector composed by two synchronous broadcaster connectors. The interface of the components are created using the library* `interface` *allowing the creation of the ports. For component ATM we have ports:*

```
arq = unsafePerformIO $ createPort
ars = unsafePerformIO $ createPort
```

The component DBRep has ports:

```
dbr = unsafePerformIO $ createPort
dbp = unsafePerformIO $ createPort
```

and, finally, the component Bank has the ports:

```
brs = unsafePerformIO $ createPort
bo  = unsafePerformIO $ createPort
brq = unsafePerformIO $ createPort
```

The main operation of a connector to orchestrate these components is the creation of two Synchronous Broadcaster, as we can see as follows,

```
bankconnector = do
                  ...
                  co1 <- createChannel (SyncBroadcaster "a" "c" "b")
                  co2 <- createChannel (SyncBroadcaster "d" "f" "e"
                  ...
```

The functional language HASKELL is a powerful tool to implement the main features of a coordination language. The library `Concurrency`, on the other hand, provides many useful devices to implement and create instances of new connectors, other than the two native channels provided by this library, (the synchronous channels and the asynchronous channel with unlimited buffering). A lot of work, in this respect, remains to be done, but this leads us to the final chapter of the thesis and its conclusions.

Chapter 8

Conclusions and Future Work

Summary

This chapter concludes the thesis and points out a number of issues for future work. A particular challenge is put forward: the development, based on the models and ideas proposed in the thesis, of a calculus of architectural patterns, to give a solid, mathematical foundation to the fundamental, but often neglected, discipline of Software Architecture. A review of some research in such a direction is included.

8.1 Concluding

The increasing demand for complex and ubiquitous applications places new challenges to the way software is designed and developed. Such challenges bring to scene a need to improve the Software Engineering discipline to cope with this new reality.

Coordination models and architectural descriptions were born within different contexts, concerns and typical application domains. However their focus is similar and recent trends in the software industry stresses the relevance of basic underlying principles. Recall, for example, the challenges entailed by the move from the *programming-in-the-large* paradigm of two decades ago, to the recent *programming-in-the-world* where not only one has to master the complexity of building and deploying a large application in time and budget, but also of managing an open-ended structure of autonomous components, possibly distributed and highly heterogeneous. Or the related shift from the traditional understanding of software as a *product* to *software as a service* [Fia04], emphasising its open, dynamic reconfigurable and evolutive structure. Terms like service *orchestration* and *choreography*, and the associated intensive research effort (see [BGG⁺05, AF04, ZXCH07, BCPV04], among many others), stress the relevance of main themes in both coordination and architectural research to modern Software Engineering. In a sense, an early definition of coordination which emphasises its goal of *finding solutions to the problem of managing the interaction among concurrent programs* [Arb98], could be taken as a main challenge to this Engineering domain.

Both, *software architecture* and *coordination models*, tackle component interaction, abstracting away the details of computation and focussing on the nature and form of the interactions. Synchronisation, communication, reconfiguration, creation and termination of computational activities are, thus, primary issues of concern.

It should also be remarked that, despite remarkable progress in the representation and use of software architecture, specification of architectural designs remain, at present, largely informal. Typically, they rely on graphical notations with poor semantics, and often limited to express only the most basic structural properties. Recent coordination models and languages, on the other hand, present a higher degree of formality — see, for example, the cases of Reo [Arb03, Arb04] or Orc [KCM06, MC07] — which stresses the case for a *coordination-driven view* of systems' architecture.

In such a context, this thesis presented a hierarchy of models to plug components together in a exogenous coordination framework. *Connectors*, modelled in chapters 2, 3 and 5, behavioural *interfaces*, addressed in chapter 4, and *configurations*, discussed in chapter 5, form the main ingredients of the proposed approach. A case study on its application to interactive systems, presented in chapter 6, and a prototyping library, documented in chapter 7, complete the thesis contributions.

In writing the thesis we were guided by the conviction that any formal model in Computer Science must provide reasonable answers to the following questions:

- How *expressive* is it (in our case, what kind of coordination schemes can be expressed within it)?
- How easy it is to *reason* within the model (to prove properties of such schemes)?
- How can it guide an effective *implementation* in the programming practice?

We still think these should be taken as a basis to assess its results.

8.2 Future work

Naturally, a lot of issues remain to be tackled.

Services and workflows. Service-oriented computing is an emerging paradigm with increasing impact on the way modern software systems are designed and developed. Services are autonomous and heterogeneous computational entities which cooperate, following a loose coupling discipline, to achieve common goals. Web services are one of the most prominent technologies in this paradigm. As an emerging technology, however, it still lacks not only sound semantical *models* but also suitable *calculi* to reason about and transform service-oriented designs.

A related topic concerns what is known in the literature as *workflow patterns* [AHKB03]. Although their role in the design of service-oriented systems is well recognized, the corresponding formalization is still a 'hot' research topic (see, *e.g.*, [AAH98, ACM04], among many others). We are currently working on their encoding in a slight extension of the formalism used here to specify behavioural interfaces. In a broader perspective, one may ask whether formal models for coordination, like the ones proposed in this thesis, can be of use in providing precise semantic foundations of emerging languages for web services composition and choreography, as, for example, WS-BPEL [WS-07] or WS-CDL [W2C05].

Mobility. Maybe the most relevant issue for future work concerns *mobility*. It is not clear how the models discussed in this thesis can be extended to cope with mobility and, in particular, with dynamic reconfiguration of connector networks. The question is, in fact, more general: we still know very little about the semantics of mobility in the context of exogenous coordination models. Tentative solutions in *e.g.*, REO [AM02] or our own contribution documented in [BB07], are still of an operational nature.

The approach discussed in chapter 3 supports dynamic reconfiguration of connections through the action of a special connector (abstracting a whole level of middleware) which manages the active possibilities of communication. The possibility of dynamic configuration of connections arises in this model from two basic assumptions: *(a)* ports have unique identifiers which can be exchanged in messages, *(b)* there is a special connector — the *orchestrator* — to manage all active connections in the network. With them mobility can be achieved in the classical *name-passing* style typical of process algebras of the π-calculus family [Mil99].

This approach should be compared with formal approaches to *dynamically reconfigurable* architectures, such as, for example, [CR97] or [WF98]. Our main current concerns, however, include the full development of the model and associated calculus, as well as its application to realistic case-studies. Moreover we intend to extend the *generic* process algebra approach discussed in chapter 4 to he π-calculus [Mil93], [Tur95]. First attempts suggest this is actually far from trivial.

Further tool support. We are also currently working on the development of a prototype implementation of the connectors model proposed in chapter 5. This will build upon and generalise the HASKELL library introduced in chapter 7. In particular, we intend to build a repository of *coordination patterns*.

Towards a calculus of architectural patterns. In retrospect, time is mature to go deep into developing a proper *theory of architectural patterns*, encompassing a semantics and a calculus, building on the lessons learnt from general research on coordination and the specific contributions of this thesis. The focus on dynamic, self-reconfigurable architectures, is relevant to a wide range of systems, from e-commerce to mobile embedded systems operated with minimal human oversight in the context of which the classical distinction between between 'development', 'deployment' and 'maintenance' tends to blur. Although a technological reality, runtime service reconfiguration is hard to model, analyse and predict. Although less common, architectures able to monitor and adapt themselves to faults (*e.g.*, lost connections or service failures), to variable resources (*e.g.*, bandwidth availability) and to unpredictable context changes, will grow in relevance in the near future.

These are big research challenges to which a *calculus of architectural patterns* would provide relevant, even if partial, answers.

The software architecture of a system is fundamental as it allows or prevents meeting the behavioral, performance and life-cycle requirements of such system. In order to facilitate the analysis and decision-making when developing/buying a system, the research community defined catalogs of architectural patterns (or styles) [BMaSS96, GS93, Gar03]. These catalogs describe, in natural language, key aspects such as contexts of application, structure, behavior, variants, consequences and known cases for relevant patterns of structural organization. The structure and behavior include components, connector types, and a set of constraints on how they can be combined. However, multiple patterns are usually applied in a single system and it becomes difficult to precisely calculate the resulting structure and behavior. Therefore, it is also hard to determine whether the desired requirements are still being met or not.

Our main proposal for a follow-up of this thesis is, therefore, the development of a rigorous discipline of architectural patterns. The envisaged discipline will consist of a calculus of such patterns and their transformations, going far beyond the common, *ad hoc* notions of architectural styles (as in classical references to architectural styles), and seeking for technology-independent formulations.

We also believe that an important issue in such a calculus would be the explicit introduction of architectural *performance* considerations. This was an issue in which we had some previous experience, in the context of the author's MSc thesis, but that was not pursued in this PhD project. We were uncertain on how such issues could be formally introduced in architectural models.

Some scientific maturity achieved along this thesis, makes us more confident in tackling this challenge. As a preliminary work on such a direction, we present in following section a review of main concepts regarding *architectural styles* (a sort of matrix for the envisaged architectural patterns) and approaches to performance in software architecture.

The otherwise strange fact that a PhD thesis ends with another research review, should just witness no thesis is a full stop, but only the beginning of new challenges.

8.3 Styles and performance in software architecture

8.3.1 Architectural styles

An architectural style consists of a vocabulary of design elements, a set of well-formedness constraints that must be satisfied by any architecture written in the style, and a semantic interpretation of the connectors. We shall now review the most popular styles documented in the literature. We believe they should be taken as the starting point for a more formal classification of architectural patterns.

Pipes and filters

In a *pipe* and *filter* style each component has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs, delivering a complete instance of the result in a standard order. This is usually accomplished by applying a local transformation to the input streams and computing incrementally so that output begins before input is consumed. Hence components are termed "filters". The connectors of this style serve as conduits for the streams, transmitting outputs of one filter to inputs of another. Hence the connectors are termed "pipes". Among the important invariants of the style, filters must be independent entities: in particular, they should not share state with other filters. Another important invariant is that filters do not know the identity of their upstream and downstream filters. Their specifications might restrict what appears on the input pipes or make guarantees about what appears on the output filters, but they may not identify the components at the ends of those pipes. Furthermore, the correctness of the output of a pipe and filter network should not depend on the order in which the filters perform their incremental processing–although fair scheduling can be assumed [AG92]. Figure 8.1 illustrates this style.

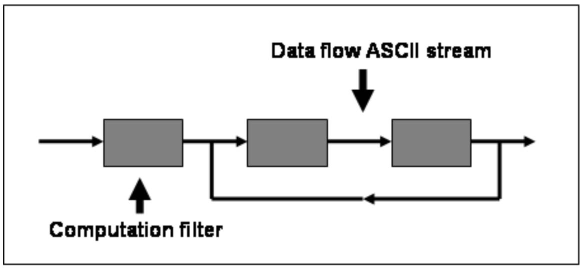


Figure 8.1: Pipes and Filter

Common specialization of this style include *pipelines*, which restrict the topologies to linear sequences of filters; bounded pipes, which restrict the amount of data that can reside on a pipe; and typed pipes, which require that the data passed between two filters have a well-defined type.

The best case of pipe and filter architectures are programs written in the Unix shell [Bac86] (see section 1.3.2 for a detailed example).

Other examples of pipes and filters occur in signal processing domains [DG90], functional programming [Kah74], and distributed systems [BWW78].

Data Abstraction and Object-Oriented Organization

In this style data representations and their associated primitive operations are encapsulated in an abstract data type or object. The components of this style are the objects – or, instances of the abstract data types. Objects are examples of a sort of component called a manager because it is responsible for preserving the integrity of a resource (here the representation). Objects interact through function and procedure invocations. Two important aspects of this style are (a) that an object is responsible for preserving the integrity of its representation (usually by maintaining some invariant over it), and (b) that the representation is hidden from other objects. Figure 8.2 illustrates this style.

Object-oriented systems have many nice properties, most of which are well

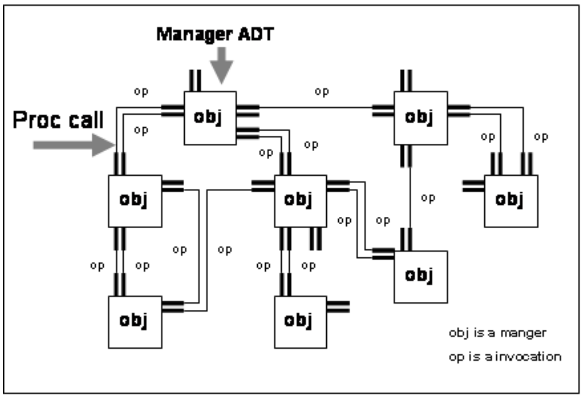


Figure 8.2: Abstract Data Types and Objects

known. Because an object hides its representation from its clients, it is possible to change the implementation without affecting those clients. Additionally, the bundling of a set of accessing routines with the data they manipulate allows designers to decompose problems into collections of interacting agents.

Event-Based

Traditionally, in a system in which the component interfaces provide a collection of procedures and functions, components interact with each other by explicitly invoking those routines. The idea behind implicit invocation is that instead of invoking a procedure directly, a component in the system can register an interest in an event by associating a procedure with the event. When the event is announced the system itself invokes all of the procedures that have been registered for the event. Thus an event announcement “implicitly” causes the invocation of procedures in other modules.

Architecturally speaking, the components in an implicit invocation style are modules whose interfaces provide both a collection of procedures (as with abstract data types) and a set of events. Procedures may be called in the usual way. But in addition, a component can register some of its procedures with events of the system. This will cause these procedures to be invoked when those events are announced at run time. Thus the connectors in an implicit invocation system includes traditional procedure call as well as bindings between event announcements and procedure calls.

Layered Systems

A layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below. In some layered systems inner layers are hidden from all except the adjacent outer layer, except for certain functions carefully selected for export. Thus in these systems the components implement a virtual machine at some layer in the hierarchy. (In other layered systems the layers may be only partially opaque.) The connectors are defined by the protocols that determine how the layers will interact. Topological constraints include limiting interactions to adjacent layers. Figure 8.3.

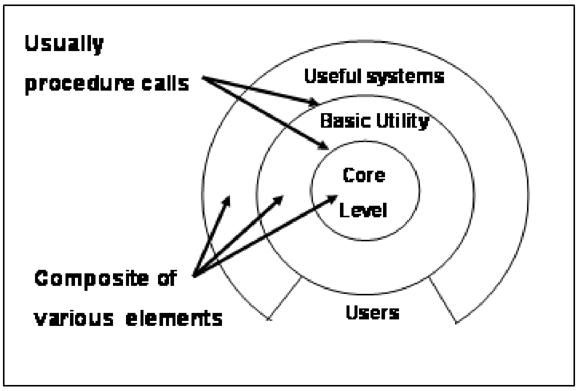


Figure 8.3: Layered Systems

The most widely known examples of this kind of architectural style are layered communication protocols [McC91].

Layered systems have several desirable properties. First, they support design based on increasing levels of abstraction. This allows implementors to partition a complex problem into a sequence of incremental steps. Second, they support enhancement. Like pipelines, because each layer interacts with at most the layers below and above, changes to the function of one layer affect at most two layers. Third, they support reuse. Like abstract data types, different implementations of the same layer can be used interchangeably, provided they support the same interfaces to their adjacent layers. This leads to the possibility of defining standard layer interfaces to which different implementors can build. (A good example is the OSI ISO model and some of the X Windows System protocols.)

Repositories

In a repository style there are two quite distinct kinds of components: a central data structure represents the current state, and a collection of independent components operate on the central data store. Interactions between the repository and its external components can vary significantly between systems.

The choice of control discipline leads to major subcategories. If the types of transactions in an input stream of transactions trigger selection of processes to execute, the repository can be a traditional database. If the current state of the central data structures is the main trigger of selecting processes to execute, the repository is called a blackboard [AG92]. (see Figure 8.4).

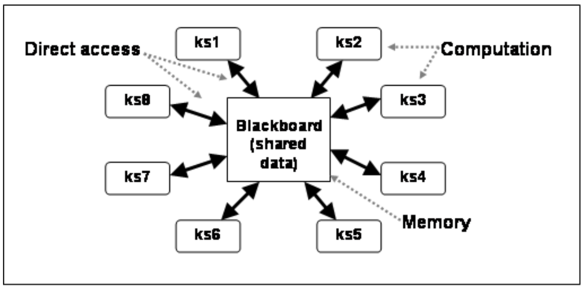


Figure 8.4: Blackboard

Table Driven Interpreters

In an interpreter organization a virtual machine is produced in software. An interpreter includes the pseudo-program being, interpreted and the interpretation engine itself. The pseudo-program includes the program itself and the interpreter's analog of its execution state (activation record). The interpretation engine includes both the definition of the interpreter and the current state of its execution. Thus an interpreter generally has four components: an interpretation engine to do the work, a memory that contains the pseudo–code to be interpreted, a representation of the control state of the interpretation engine, and a representation of the current state of the program being simulated. (See Figure 8.5).

Heterogeneous Architectures

Thus far we have been speaking primarily of "pure" architectural styles. While it is important to understand the individual nature of each of these styles, most systems

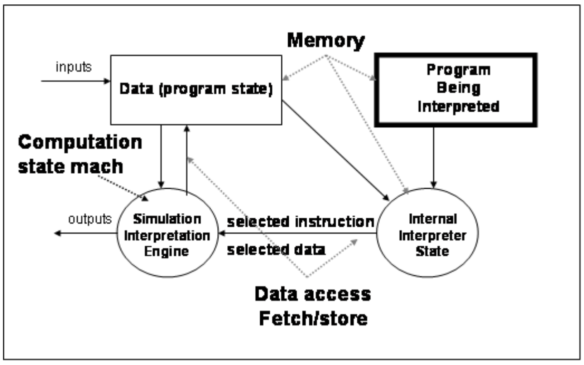


Figure 8.5: Interpreter

typically involve some combination of several styles.

There are different ways in which architectural styles can be combined. One way is through hierarchy. A component of a system organized in one architectural style may have an internal structure that is developed a completely different style. For example, in a Unix pipeline the individual components may be represented internally using virtually any style – including, of course, another pipe and filter, system.

What is perhaps more surprising is that connectors, too, can often be hierarchically decomposed. For example, a pipe connector may be implemented internally as a FIFO queue accessed by insert and remove operations.

A second way for styles to be combined is to permit a single component to use a mixture of architectural connectors. For example, a component might access a repository through its interface, but interact through pipes with other components in a system, and accept control information through another part os its interface. (In fact, Unix pipe and filter systems do this, the file system playing the role of the repository and initialization switches playing the role of control.)

Another example is an “active database”. This is a repository which actives external components through implicit invocation. In this organization external components register interest in portions of the database. The database automatically invokes the appropriate tools based on this association. (Blackboards are often constructed this way; knowledge sources are associated with specific kinds of data, and are activated whenever that kind of data is modified.)

8.3.2 Performance in software architecture

There are different approaches and methodologies concerning the derivation of performance models from software architecture specification. In this section some of these methodologies will be considered.

Each approach is based on a certain type of performance model and specification language such as process algebras ([GUHR93], [Hil96]), Petri nets ([MBC$^+$95], [Mol82]) and Chemichal abstract machine [IW95], and UML based specification [BRJ99]. Performance models include queueing networks (QN) and their extensions called Extended Queueing Networks (EQN) and Layered Queueing Networks (LQN), Stochastic Timed Petri nets (STPN), Stochastic Process Algebras (SPA) and simulation models. Some of the proposed methods are based on the Software Performance Engineering (SPE) methodology introduced by Smith in [Sim90].

Software Performance Engineering has been the first comprehensive approach to the integration of performance analysis into the software development process, from the earliest stages to the end. The SPE methodology is based on two models:

- Software execution model that is based on execution graphs (EG) and represents the software execution behavior, and
- System execution model that is based on queueing network models and represents the computer system platform, including hardware and software components.

The analysis of the software model gives information concerning the resource requirements of the software system. The obtained results, together with information about the hardware devices, are the input parameters of the system execution model, which represents the model of the whole software/hardware system.

Methodologies

Williams and Smith [LW98] apply the Software Performance Engineering methodology (SPE) to evaluate the performance characteristics of a software architecture. The emphasis is in the construction and analysis of the software execution model, which is considered the target model of the specified SA and is obtained from the Sequence diagrams. The Class and Deployment diagrams contribute to complete the description of the SA, but are not involved in the transformation process. The SPE process requires additional information that includes software resource requirements for processing steps and computer configuration data.

In [DM98], Menascè and Gomaa present a methodology to derive QN performance models from SA specification. It has been developed and used by the authors in a design of client/server applications. The methodology is based on CLISSPE (CLIent/Server Software Performance Evaluation) [Men97], a language for the software performance engineering of client/server applications. Although the methodology does not explicitly use UML, the functional requirement of the system are specified in terms of use cases, and the system model is specified by the analogous of a Class diagram. The use cases, together with the client/server SA specification and the mapping associating software components to hardware devices, are used to develop a CLISSPE program specification. The CLISSPE system provides a compiler that generates a corresponding QN model. By considering specific scenarios one can define the QN parameters and apply appropriate solution methods, such as the Layered Queuing Models (LQN) [RS95], [WNPM95] to obtain performance results.

Balsamo et alt. [BIM98] provide a method for the automatic derivation of a queuing network model from a SA specification, described using the CHAM formalism (CHemical Abstract Machine) [IW95]. Informally, the CHAM specification of a SA is given by a set of molecules which represent the static components of the architecture, a set of reaction rules which describe the dynamic evolution of the system through reaction steps, and an initial solution which describes the initial static configuration of the system. In [IW95] is presented an algorithm to derive a QN model from the CHAM specification of a SA architecture. It is based on the analysis of the Labeled Transition System (LTS) that represents the dynamic behavior of the CHAM architecture, and that can be automatically derived from the CHAM specification. The algorithm does not completely define the QN model whose parameters, such as the service time distributions and the customer's arrival processes, have to be specified by the designer. The solution of the QN model is derived by analytical methods or possibly by symbolic evaluation. Parameter instantiation identify potential implementation scenarios and the performance results allow to provide insights on how to carry on the development process in order to satisfy given performance criteria.

Cortellessa and Mirandola [VC00] propose a methodology making a joint use of information from different UML diagrams to generate a performance model of the specified system. They refer to SPE methodology and specifie the software architecture by using Deployment, Sequence, and Use Case diagrams. This approach is a more formal extension of the WS approach [LW98] and consists of the following steps:

Andolfi et alt. [AABI00] propose an approach to automatically generate queuing network models from software architecture specifications described by means of Message Sequence Charts (MSC), that correspond to Sequence diagrams in the UML terminology. The idea is to analyze MSCs in terms of the trace languages (sequences of events) they generate, in order to single out the real degree of parallelism among components and their dynamic dependencies. This information is then used to build a QN model corresponding to the software architecture description. The authors present an algorithm to perform this step. This approach is built on the previous work [BIM98] to overcome the drawback of the high computational complexity due to possible state space explosion of the finite state model of the CHAM description.

Aquilani [ABI01] proposes an approach which concerns the derivation of QN models from Labeled Transition Systems (LTS) describing the dynamic behavior of SAs. Starting from a LTS description of a SA makes it possible to abstract from any particular SA specification language. The approach assumes that LTSs are the only knowledge on the system that they can use. This means, in particular, that it does not use any information concerning the system implementation or deployment.

Bernardo et alt. [MB00] propose an architectural description language based on stochastically timed Process Algebras. This approach provides an integration of a formal specification language and performance models. The aim is to describe and analyze both functional and performance properties of SAs in a formal framework. The approach proposes the adoption of an architectural description language called ÆMPA, gives its syntax with a graphical and textual notation and its semantics in terms of EMPA specifications, that is a stochastically timed process algebra [Ber99]. The authors illustrate various functional and non-functional properties, including performance evaluation which is based on the generation of the underlying Markov chain that is numerically solved. To this aim the authors propose the use of TwoTowers [Ber99], a software tool for systems modeling and analysis of functional and performance properties, that support system EMPA description.

Methodologies based on architectural patterns

Architectural patterns identify frequently used architectural solutions and are used to describe SAs. Each pattern is described by its structure (what are the components) and its behavior (how they interact). Some approaches consider software specification of architectural patterns and derive their corresponding performance models. They use UML specification. The approaches identify a direct correspondence between each pattern and its performance model, which can be immediately derived.

Gomaa and Menascè in [GM00] investigate the design and performance modeling of component interconnection patterns for client/server systems. Such patterns define and encapsulate the way client and server components of software architecture communicate with each other via connectors. The idea is to start with UML design models of component interconnection patterns, using Class diagrams (to model their static aspects) and Collaboration diagrams (to depict the dynamic interactions between components and connectors objects - instances of the classes depicted on the Class diagrams). Such models are then provided with additional performance annotations, and translated into an XML notation, in order to capture both the architecture and performance parameters in one notation. The performance models of the considered patterns are extended QN and their definition, based on previous work of the authors, depends on the type of communication. The EQN model solution is obtained by Markov chain analysis or approximate analytical methods.

In ([PW99], [Pet00]) Petriu and Wang consider a significant set of architectural patterns (pipe and filters, client/server, broker, layers, critical section and master-slave) specified by using UML-Collaborations that are combined Class and Sequence diagrams showing explicitly the collaborating objects. The approach shows the corresponding performance models based on LQN models. Moreover, they propose a systematic approach to build performance models of complex SAs based on combinations of the considered patterns. The approach follows the SPE methodology and generates the software and system execution models by applying graph transformation techniques. SAs are specified using UML-Collaborations, Deployment and Use Case diagrams. The Sequence diagram part of the UML-Collaboration is used to obtain the software execution model (which is represented as a UML Activity diagram); the Class part is used to obtain the system execution model (which is represented as a LQN model). Use Case diagrams provide information on the workloads, and Deployment diagrams allow for the allocation of software components to hardware sites.

Simulation methods

Arief and Speirs [LA00] have proposed an approach that present a simulation framework named Simulation Modelling Language (SimML) to generate a simulation program from the system design specified with the UML. The proposed UML tool allows the user to draw Class and Sequence diagrams and to specify the information needed for the automatic generation of the process oriented simulation model. The simulation program is generated in the Java programming language. The approach proposes an XML translation of the specified UML models, in order to store the information about the design and the simulation data in a structured way.

The approach proposed by de Miguel et alt. [MLH$^+$00] focus on real time systems, and proposes extensions of UML diagrams to express temporal requirements and resource usage. The extension is based on the use of stereotypes, tagged values and stereotyped constraints. SAs are specified using the extended UML diagrams without restrictions on the type of diagrams to be used. Then these UML diagram are used as input for the automatic generation of the corresponding simulation models in OPNET. They also define a middleware model for scheduling analysis. The simulation model is defined by instantiating with the application information the generic models that represent the various UML metaclasses. The approach generates submodels for each application element and combines them into a unique simulation model of the UML application. The approach provides also a feedback mechanism: after the model has been analyzed and simulated, some results are included in the tagged values. This constitutes a relevant feature, which ease the SA designer in obtaining feedback from the performance evaluation results.

Case studies

Some approaches present the generation of performance models from a software specification through an example or a case study. They consider UML specification and different types of performance models.

In [PK99] King and Pooley show, through an example, how to generate Stochastic Timed Petri Net models from the UML specification of systems. They consider Use Case diagrams and combined diagrams consisting of a Collaboration diagram with State diagrams (i.e. statecharts) of all the collaborating objects embedded within them. The idea is to translate each State diagram that represents an object of the Collaboration diagram into a Petri net: states and transitions in the State diagram are represented by places and transitions in the Petri net, respectively. The obtained Petri nets can be combined to obtain a unique STPN model that represents the whole system, specifically a Generalized stochastic Petri net (GSPN) [MA86]. The merging of nets is only explained via the running example, i.e. the thesis does not include a general merging procedure. The GSPN can be analyzed by specific tools such as the SPNP package.

In [RP99] Pooley and King describe some preliminary ideas on how to derive a queuing network model from the UML specification of a system. Use Case diagrams are used to specify the workloads and the various classes of requests of the system being modeled. Implementation diagrams are used to define contention and to quantify the available system resources. The idea is to define a correspondence between combined Deployment and Component diagrams and queuing network models, by mapping components and links to service centers.

In [Poo99] Pooley describes how to derive stochastic process algebra models from UML specifications. More precisely, the starting point is, like in [PK99], the specification of a system via a combined diagram consisting of a Collaboration diagram with State diagrams (i.e. statecharts) of all the collaborating objects embedded within them. The idea is to produce a Stochastic Process Algebra description of each object of the Collaboration diagram and to combine them into a unique model. The thesis shows how this can be done on a real although simple example. The presents also an attempt to generate a continuous-time Markov chain directly from the combined UML diagram of the running example. The key observation here is that, at any time, each object of the collaboration diagram must be in one, and only one, of its internal states. The combination of the objects current states is called "marking”. The idea is to derive all possible markings by following through the interactions: this allows building the corresponding state transition diagram and, then, the underlying Markov chain.

Smith and Williams in [SW97] present an example to illustrate the derivation of a performance model from an object-oriented design model, and propose the use of the SPE•ED tool that supports the SPE methodology to evaluate object-oriented systems. Starting from a set of scenarios described by Message Sequence Charts, they derive the execution graphs that define the software execution model and then by the analyst specification of computer resource requirements they define the system execution model. The QN model is analyzed by approximate analytical methods or by simulation integrated in the SPE•ED tool.

In [Hoe00] Hoeben discusses some rules that can be used to express or add information useful to derive performance evaluation from the various UML diagrams. The work proposes some UML extensions based on the use of stereotypes and tagged values and some rules to propagate user requests specified by UML models to define the performance model. These rules allow performance evaluation of UML models at various levels of abstraction and a prototype tool to automatically create performance estimates based on QN models uses them.

Bibliography

[AABI00] F. Andolfi, F. Aquilani, S. Balsamo, and P. Inverardi. Deriving performance models of software architectures from message sequence charts. WOSP2000, 2000.

[AAH98] N. R. Adam, V. Atluri, and Wei-Kuang Huang. Modeling and analysis of workflows using petri nets. *J. Intell. Inf. Syst.*, 10(2):131–158, 1998.

[ABI01] F. Aquilani, S. Balsamo, and P. Inverardi. Performance analysis at the software architectural design level. *In Performance Evaluation*, 45:205–221, 2001.

[ACG86] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, 1986.

[ACKM04] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services — Concepts, Architectures and Applications*. Data-centric Systems and Applications. Springer-Verlag, 2004.

[ACM04] Roberta Amici, Flavio Corradini, and Emanuela Merelli. A process algebra view of coordination models with a case study in computational system biology. In L. Bocchi and P. Ciancarini, editors, *Proceedings of the First International Workshop on Petri Nets and Coordination (PNC04), Satellite Event of the 25th International Conference on Application and Theory of Petri Nets, Bologna, Italy, June 21, 2004*, pages 33–47, 2004.

[Acz88] P. Aczel. *Non-Well-Founded Sets*. CSLI Lecture Notes (14), Stanford, 1988.

[Acz93] P. Aczel. Final universes of processes. In Brooks et al, editor, *Proc. Math. Foundations of Programming Semantics*. Springer Lect. Notes Comp. Sci. (802), 1993.

[AF04] L. F. Andrade and J. L. Fiadeiro. Composition contracts for service interaction. *Journal of Universal Computer Science*, 10(4):751–761, 2004.

[AG92] R. Allen and D. Garlan. A formal approach to software architectures. *In Proceedings of IFIP'92 (J. van Leewen, ed.)*, 1992.

[AG97] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM TOSEM*, 6(3):213–249, 1997.

[AH01] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC/FSE-9: Proceedings of the 8th European Software Engineering Conference*, pages 109–120, New York, NY, USA, 2001. ACM Press.

[AHDLM96] J-M Andreoli, C. Hankin, and Eds. D. Le Métayer. Coordination programming: Mechanisms, models and semantics. Imperial College Press, 1996.

[AHKB03] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.

[AHS90] J. Adamek, H. Herrlich, and G. E. Strecker. *Abstract and Concrete Categories*. John Wiley & Sons, Inc (revised electronic edition in 2004), 1990.

[AHS93] F. Arbab, I. Herman, and P. Spilling. An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, 5(1):23–70, 1993.

[ALSN01] Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Piccola – a small composition language. In Howard Bowman and John Derrick, editors, *Formal Methods for Distributed Processing – A Survey of Object-Oriented Approaches*, pages 403–426. Cambridge University Press, 2001.

[AM02] F. Arbab and F. Mavadatt. Coordination through channel composition. In *Proc. Coordination Languages and Models*. Springer Lect. Notes Comp. Sci. (2315), 2002.

[AR02] F. Arbab and J. Rutten. A coinductive calculus of component connectors. CWI Tech. Rep. SEN-R0216, CWI, Amsterdam, 2002.

[AR03] F. Arbab and J. J. M. M. Rutten. A coinductive calculus of component connectors. In Martin Wirsing, Dirk Pattinson, and Rolf Hennicker, editors, *Recent Trends in Algebraic Development Techniques, 16th Inter. Workshop, WADT 2002, Revised Selected Papers*, pages 34–55. Springer Lect. Notes Comp. Sci. (2755), 2003.

[Arb96] Farhad Arbab. The iwim model for coordination of concurrent activities. In Paolo Ciancarini and Chris Hankin, editors, *Proc. Coordination Languages and Models, First Inter. Conf., COORDINATION '96, Cesena, Italy, April 15-17*, volume 1061, pages 34–56. Springer Lect. Notes Comp. Sci. (1061), 1996.

[Arb98] Farhad Arbab. What do you mean, coordination. In *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)*, 1998.

[Arb02] F. Arbab. A channel-based coordination model for component composition. CWI Tech. Rep. SEN-R0203, CWI, Amsterdam, 2002.

[Arb03] F. Arbab. Abstract behaviour types: a foundation model for components and their composition. In F. S. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. First International Symposium on Formal Methods for Components and Objects (FMCO'02)*, pages 33–70. Springer Lect. Notes Comp. Sci. (2852), 2003.

[Arb04] F. Arbab. Reo: a channel–based coordination model for component composition. *Mathematical Structures in Comp. Sci.*, 14(3):329–366, 2004.

[Bac86] M. J. Bach. *The Design of the UNIX Operating System*, volume ch. 5.12 of *Software Series*. Prentice–Hall, 1986.

[Bac88] R. Backhouse. An exploration of the Bird-Meertens formalism. CS 8810, Groningen University, 1988.

[Bac02] R. Backhouse. Galois connections and fixed point calculus. In R. Crole, R. Backhouse, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Constuction*, pages 89–148. Springer Lect. Notes Comp. Sci. (2297), 2002.

[Bar00] L. S. Barbosa. Components as processes: An exercise in coalgebraic modeling. In S. F. Smith and C. L. Talcott, editors, *FMOODS'2000 - Formal Methods for Open Object-Oriented Distributed Systems*, pages 397–417. Kluwer Academic Publishers, September 2000.

[Bar01] L. S. Barbosa. Process calculi *à la* Bird-Meertens. In Marina Lenisa Andrea Corradini and Ugo Montanari, editors, *CMCS'01*, volume 44.4, pages 47–66, Genova, April 2001. Elect. Notes in Theor. Comp. Sci., Elsevier.

[Bar03] L. S. Barbosa. Towards a Calculus of State-based Software Components. *Journal of Universal Computer Science*, 9(8):891–909, August 2003.

[BB04a] M. A. Barbosa and L. S. Barbosa. A relational model for component interconnection. *Journal of Universal Computer Science*, 10(7):808–823, 2004.

[BB04b] M. A. Barbosa and L. S. Barbosa. Specifying software connectors. In K. Araki and Z. Liu, editors, *1st International Colloquium on Theorectical Aspects of Computing (ICTAC'04)*, pages 53–68, Guiyang, China, September 2004. Springer Lect. Notes Comp. Sci. (3407).

[BB06] M. A. Barbosa and L. S. Barbosa. Configurations of web services. In *FOCLASA'06: Proc. 5th Inter. Workshop on the Foundations of Coordination Languages and Software Architectures*, volume 175 (2) of *Electronic Notes in Theoretical Computer Science*, pages 39–57. Elsevier, 2006.

[BB07] M. A. Barbosa and L. S. Barbosa. An orchestrator for dynamic interconnection of software components. In *Proc. 2nd International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord'06)*, volume 181, pages 49–61, Bologna, Italy, June 2007. Elsevier.

[BB09] M. A. Barbosa and L. S. Barbosa. A perspective on service orchestration. *Science of Computer Programming*, 74(9):671–687, 2009.

[BBC07] M. A. Barbosa, L. S. Barbosa, and J. C. Campos. Towards a coordination model for interactive systems. In A. Cerone and P. Curzon, editors, *FMIS 2007: Proc. 1st Inter. Workshop in Formal Methods for Interactive Systems*, volume 347 of *Electronic Notes in Theoretical Computer Science*, pages 89–103. Elsevier, 2007.

[BBC09] Marco A. Barbosa, L. S. Barbosa, and José C. Campos. A coordination codel for interactive components. In F. Arbab and M. Sirjani, editors, *Proc. of FSEN 2009, Kish, Iran*. Springer Lect. Notes Comp. Sci. (to appear), 2009.

[BCG97] Robert Bjornson, Nicholas Carriero, and David Gelernter. From weaving threads to untangling the web: A view of coordination from linda's perspective. In David Garlan and Daniel Le Metayer, editors, *Proc. of Second Inter. Conf. on Coordination Languages and Models, COORDINATION '97, Berlin, Germany*, pages 1–17. Springer Lect. Notes Comp. Sci. (1282), 1997.

[BCK03] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice (2nd ed.)*. Addison-Wesley, 2003.

[BCPV04] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. Formalizing web services choreographies. In *Proc. First Inter. Workshop on Web Services and Formal Methods*, volume 105, pages 73–94, Pisa, Italy, 2004.

[Ber99] Marco Bernardo. Lets evaluate performance algebraically. *ACM Computing Surveys (CSUR)*, 31(3es):7, 1999.

[BGG$^+$05] N. Busi, R. Gorrieri, C. Guidi, R. Luchi, and G. Zavattaro. Choreography and orchestration: A synergic approach for systems design. In B. Benatallah, F. Casati, and P. Traverso, editors, *Proc. ICSOC 2005 Thrid Inter. Conf. on Service-Oriented Computing*, pages 228–240, Amsterdam, The Netherlands, 2005.

[BGR$^+$99] Klaus Bergner, Radu Grosu, Andreas Rausch, Alexander Schmidt, Peter Scholz, and Manfred Broy. Focusing on mobility. In *HICSS*, 1999.

[BH93] R. C. Backhouse and P. F. Hoogendijk. Elements of a relational theory of datatypes. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, pages 7–42. Springer Lect. Notes Comp. Sci. (755), 1993.

[BIM98] S. Balsamo, P. Inverardi, and C. Mangano. An approach to performance evaluation of software architectures. Workshop on Software and Performance, WOSP'98, pages 12–16, 1998.

[Bir87] R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F*, pages 3–42. Springer-Verlag, 1987.

[Bir98] R. Bird. *Functional Programming Using Haskell*. Series in Computer Science. Prentice-Hall International, 1998.

[BJJM98] R. C. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming: An introduction. In S. D. Swierstra, P. R. Henriques, and J. N. Oliveira, editors, *Third International Summer School on Advanced Functional Programming, Braga*, pages 28–115. Springer Lect. Notes Comp. Sci. (1608), September 1998.

[BM87] R. S. Bird and L. Meertens. Two exercises found in a book on algorithmics. In L. Meertens, editor, *Program Specification and Transformation*, pages 451–458. North-Holland, 1987.

[BM97] R. Bird and O. Moor. *The Algebra of Programming*. Series in Computer Science. Prentice-Hall International, 1997.

[BM07] J. K. F. Bowles and S. Moschoyiannis. Concurrent logic and automata combined: A semantics for components. In C. Canal and M. Viroli, editors, *Proc. of FOCLASA'06*, volume 175 (2), pages 135–151. Elsevier, 2007.

[BMaSS96] Frank Buschmann, Regine Meunier, Hans Rohnert andPeter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996.

[BNP03] Rémi Bastide, David Navarre, and Philippe A. Palanque. A tool-supported design framework for safety critical interactive systems. *Interacting with Computers*, 15(3):309–328, 2003.

[BO02] L. S. Barbosa and J. N. Oliveira. Coinductive interpreters for process calculi. In *Proc. of FLOPS'02*, pages 183–197, Aizu, Japan, September 2002. Springer Lect. Notes Comp. Sci. (2441).

[BO03] L. S. Barbosa and J. N. Oliveira. State-based components made generic. In H. Peter Gumm, editor, *CMCS'03, Elect. Notes in Theor. Comp. Sci.*, volume 82.1. Elsevier, 2003.

[BO06] L. S. Barbosa and J. N. Oliveira. Transposing partial components: an exercise on coalgebraic refinement. *Theor. Comp. Sci.*, 365(1-2):2–22, 2006.

[BOS08] L. S. Barbosa, J. N. Oliveira, and A. M. Silva. Calculating invariants as coreflexive bisimulations. In J. Meseguer and G. Rosu, editors, *Algebraic Methodology and Software Technology, 12th International Conference, AMAST 2008, Urbana, IL, USA, July 28-31, 2008, Proceedings*, pages 83–99. Springer Lect. Notes Comp. Sci. (5140), 2008.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[BRS+00] K. Bergner, A. Rausch, M. Sihling, A. Vilbig, and M. Broy. A Formal Model for Componentware. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 189–210. Cambridge University Press, 2000.

[BSAR06] C. Baier, M. Sirjani, F. Arbab, and J. J. M. M. Rutten. Modeling component connectors in reo by constraint automata. *Science of Computer Programming*, 61(2):75–113, 2006.

[BTL05] A. R. Du Bois, P. Trinder, and H. Loidl. mHaskell: Mobile computation in a purely functional language. *Journal of Universal Computer Science*, 11(7):1234–1254, 2005.

[BWW78] M. R. Barbacci, C. B. Weinstock, and J. M. Wing. Programming at the processor–memory–switch level. In *In Proceedings of the 10th International Conference on Software Engineering (Singapore)*, pages 19–28. IEEE Computer Society Press, 1978.

[CBO05] A. Cruz, L. Barbosa, and J. Oliveira. From algebras to objects: Generation and composition. *Journal of Universal Computer Science*, 11(10):1580–1612, 2005.

[CCA07] D. Clarke, D. Costa, and F. Arbab. Connector colouring I: Synchronisation and context dependency. *Science of Computer Programming*, 66(3):205–225, 2007.

[CF92] R. Cockett and T. Fukushima. About Charity. Yellow Series Report No. 92/480/18, Dep. Computer Science, University of Calgary, June 1992.

[CH96] Paolo Ciancarini and Chris Hankin, editors. *Coordination Languages and Models, First International Conference, COORDINATION '96, Cesena, Italy, April 15-17, 1996, Proceedings*, volume 1061 of *Lecture Notes in Computer Science*. Springer, 1996.

[CH01] J. C. Campos and M. D. Harrison. Model checking interactor specifications. *Automated Software Engineering*, 8(3/4):275–310, August 2001. ISSN: 0928-8910.

[CH08] J. C. Campos and M. D. Harrison. Systematic analysis of control panel interfaces using formal tools. In *XVth International Workshop on the Design, Verification and Specification of Interactive Systems (DSV-IS 2008)*, pages 72–85. Springer Lect Notes in Comp. Sci. (5136), July 2008.

[Cos10] D. Costa. *Formal models for context dependent connectors for distributed software components and services (forthcoming PhD thesis)*. PhD thesis, Vrije Universiteit Amsterdam, 2010.

[CR97] G. Costa and G. Reggio. Specification of abstract dynamic data types: A temporal logic approach. *Theor. Comp. Sci.*, 173(2), 1997.

[CS92] R. Cockett and D. Spencer. Strong categorical datatypes I. In R. A. G. Seely, editor, *Proceedings of Int. Summer Category Theory Meeting, Montréal, Québec, 23–30 June 1991*, pages 141–169. AMS, CMS Conf. Proceedings 13, 1992.

[CS95] R. Cockett and D. Spencer. Strong categorical datatypes II: A term logic for categorical programming. *Theor. Comp. Sci.*, 139:69–113, 1995.

[DF05] Anke Dittmar and Peter Forbrig. A unified description formalism for complex hci-systems. In *SEFM '05: Proc. 3rd IEEE Inter. Conf. on Software Engineering and Formal Methods*, pages 342–351. IEEE Computer Society, 2005.

[DG90] N. Delisle and D. Garlan. Applying formal specification to industrial problems: A specification of an oscilloscope. *IEEE Software*, 1990.

[DH93] David J. Duke and Michael D. Harrison. Abstract interaction objects. *Computer Graphics Forum*, 12(3):25–36, 1993.

[DM98] H. Gomaa D.A. Menascè. On a language based method for software performance engineering of client/server systems. Proc. of WOSP'98, Santa Fe, New Mexico, USA, pages 63–69, 1998.

[dSDR98] Bruno d'Ausbourg, Christel Seguin, Guy Durrieu, and Pierre Roché. Helping the automated validation process of user interfaces systems. In *ICSE '98: Proc. 20th Inter. Conf. on Software Engineering*, pages 219–228. IEEE Computer Society, 1998.

[Fia04] J. L. Fiadeiro. Software services: scientific challenge or industrial hype? In K. Araki and Z. Liu, editors, *Proc. First International Colloquim on Theoretical Aspects of Computing (ICTAC'04), Guiyang, China*, pages 1–13. Springer Lect. Notes Comp. Sci. (3407), 2004.

[Fok92a] M. M. Fokkinga. Calculate categorically! *Formal Aspects of Computing*, 4(4):673–692, 1992.

[Fok92b] M.M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992.

[FP90] G. Faconti and F. Paternò. An approach to the formal specification of the components of an interaction. In C. Vandoni and D. Duce, editors, *Eurographics '90*, pages 481–494. North-Holland, 1990.

[GAO95] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Proc. 17th Inter. Conf. on Software Enginneering, ACM SIGSOFT*, pages 179–185, 1995.

[Gar03] D. Garlan. Formal modeling and analysis of software architecture: Components, connectors and events. In M. Bernardo and P. Inverardi, editors, *Third International Summer School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures (SFM 2003)*. Springer Lect. Notes Comp. Sci, Tutorial, (2804), Bertinoro, Italy, September 2003.

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.

[Gib97] J. Gibbons. Conditionals in distributive categories. CMS-TR-97-01, School of Computing and Mathematical Sciences, Oxford Brookes University, 1997.

[GM97] David Garlan and Daniel Le Métayer, editors. *Coordination Languages and Models, Second International Conference, COORDINATION '97, Berlin, Germany, September 1-3, 1997, Proceedings*, volume 1282 of *Lecture Notes in Computer Science*. Springer, 1997.

[GM00] J. Goguen and G. R. Malcolm. A hidden agenda. *Theor. Comp. Sci.*, 245(1):55–101, 2000.

[GS93] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering (volume I)*. World Scientific Publishing Co., 1993.

[GS04] V. Gruhn and C. Schäfer. An architecture description language for mobile distributed systems. In Ron Morrison Flavio Oquendo, Brian Warboys, editor, *Software Architecture - Proceedings of the First European Workshop, EWSA 2004*, pages 212–218. Springer-Verlag, 2004.

[GUHR93] N. Götz, U U. Herzog, and M. Rettelbach. Multiprocessor and distributed system design: The integration of functional specification and performance analysis using stochastic process algebra. In *Proc. of the 16th Int. Symp. on computer Performance Modelling, Measurement and Evaluation (PERFORMANCE 1993)*, volume 729 of *Springer LNCS*, pages 121–146, 1993.

[Hag87a] T. Hagino. *Category Theoretic Approach to Data Types*. Ph.D. thesis, tech. rep. ECS-LFCS-87-38, Laboratory for Foundations of Computer Science, University of Edinburgh, UK, 1987.

[Hag87b] T. Hagino. A typed lambda calculus with categorical type constructors. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *Category Theory and Computer Science*, pages 140–157. Springer Lect. Notes Comp. Sci. (283), 1987.

[Hil96] J. Hillston. *A Compositional Approach to Performance Modeling*. Cambridge University Press, 1996.

[HM85] M. C. Hennessy and A. J. R. G. Milner. Algebraic laws for non-determinism and concurrency. *Journal of ACM*, 32(1):137–161, 1985.

[Hoa85] C. A. R Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, 1985.

[Hoe00] F. Hoeben. Using uml models for performance calculation. WOSP2000, pages 77–82, 2000.

[HT90] M. Harrison and H. Thimbleby, editors. *Formal Methods in Human-Computer Interaction*. Cambridge Series on Human-Computer Interaction. Cambridge University Press, 1990.

[IBM03] IBM. Web services architecture overview: the next stage of evolution for e-business. `http://www.ibm.com/developerworks/web/library/w-ovr/`, 2003.

[IW95] Paola Inverardi and Alexander L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, 21(4), 1995.

[Jac96] B. Jacobs. Objects and classes, co-algebraically. In C. Lengauer B. Freitag, C.B. Jones and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Academic Publishers, 1996.

[Jac99] B. Jacobs. The temporal logic of coalgebras via Galois algebras. Techn. rep. CSI-R9906, Comp. Sci. Inst., University of Nijmegen, 1999.

[Jac02] Bart Jacobs. Exercises in coalgebraic specification. In R. Backhouse, R. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, pages 237–280. Springer Lect. Notes Comp. Sci. (2297), 2002.

[JGF96] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.

[JMA96] Daniel Le Metayer Jean-Marc Andreoli, Chris Hankin. *Coordination Programming: Mechanisms, Models, and Semantics*. Imperial College Press, 1996.

[JOT06] E. B. Johnsen, O. Owe, and A. B. Torjusen. Validating behavioural component interfaces in rewriting logic. volume 159, pages 187–204. Elsevier, 2006.

[JR97] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–159, 1997.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 1974.

[KCM06] David Kitchin, William R. Cook, and Jayadev Misra. A language for task orchestration and its semantic properties. In Christel Baier and Holger Hermanns, editors, *Proc. 17th Inter. Conf. Concurrency Theory, CONCUR 2006, Bonn, Germany, August 27-30*, pages 477–491. Springer Lect. Notes Comp. Sci. (4137), 2006.

[Kie98a] R. B. Kieburtz. Codata and comonads in HASKELL. Unpublished manuscript, 1998.

[Kie98b] R. B. Kieburtz. Reactive functional programming. In David Gries and Willem-Paul de Roever, editors, *Programming Concepts and Methods (PROCOMET'98)*, pages 263–284. Chapman and Hall, Junho 1998.

[Kir02] Zeliha Dilsun Kirli. *Mobile Computation with Functions*, volume 5 of *Advances in Information Security*. Springer, 2002.

[Koc72] A. Kock. Strong functors and monoidal monads. *Archiv für Mathematik*, 23:113–120, 1972.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theor. Comp. Sci.*, (27):333–354, 1983.

[LA00] N.A. Speirs L.B. Arief. A uml tool for an automatic generation of simulation programs. WOSP2000, pages 71–76, 2000.

[LF02] A. Lopes and L. Fiadeiro. On how distribution and mobility interfere with coordination. In *Proc. of WADT*, pages 343–358. Springer Lect. Notes Comp. Sci (2755), 2002.

[Lum99] M. Lumpe. *A π-calculus Based Approach to Software Composition*. PhD thesis, University of Bern, January 1999.

[LW98] C.U. Smith L.G. Williams. Performance evaluation of software architectures. in Proc. of WOSP'98, Santa Fe, New Mexico, USA, pages 164–177, 1998.

[MA86] G. Conte M. Ajmone, G. Balbo. *Performance Models of Miltiprocessor Performance*. MIT Press, 1986.

[Mac71] S. Mac Lane. *Categories for the Working Mathematician*. Springer Verlag, 1971.

[Mal90] G. R. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–279, 1990.

[Mar95] P. Markopoulos. On the expression of interaction properties within an interactor model. In *In P. Palanque and R. Bastide (eds.), Design, Specification and Verification of Interactive Systems'95*, 1995.

[MB00] L. Donatiello M. Bernardo, P. Ciancarini. Aempa: A process algebraic description language for the performance analysis of software architectures. WOSP2000, 2000.

[MB04] Sun Meng and L. S. Barbosa. On refinement of generic software components. In C. Rettray, S. Maharaj, and C. Shankland, editors, *10th Int. Conf. Algebraic Methods and Software Technology (AMAST)*, pages 506–520, Stirling, 2004. Springer Lect. Notes Comp. Sci. (3116). Best Student Co-authored Paper Award.

[MB05] Sun Meng and L. S. Barbosa. Components as coalgebras: The refinement dimension. *Theor. Comp. Sci.*, 351:276–294, 2005.

[MBC$^+$95] Marco Marsan, Gianfranco Balbo, Gianni Conte, S Donatelli, and G Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley and Sons, Chichester England, 1995.

[MC07] J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. *Software and System Modeling*, 6(1):83–110, 2007.

[McC91] G. R. McClain. *Open Systems Interconnection Handbook*. NY:Intertext Publications. McGraw–Hill Book Company, 1991.

[McL92] C. McLarty. *Elementary Categories, Elementary Toposes*, volume 21 of *Oxford Logic Guides*. Clarendon Press, 1992.

[Mee92] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.

[Men97] D.A. Menascè. A framework for software performance engineering of client/server systems. Proc. of the 1997 Computer Measurement Group Conference, Orlando, Florida, 1997.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 1991 ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144. Springer Lect. Notes Comp. Sci. (523), 1991.

[Mil89] R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice-Hall International, 1989.

[Mil93] R. Milner. The polyadic pi-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.

[Mil99] R. Milner. *Communicating and Mobile Processes: the π-Calculus*. Cambridge University Press, 1999.

[MKG99] Jeff Magee, Jeff Kramer, and Dimitra Giannakopoulou. Behaviour analysis of software architectures. In *WICSA1: Proc. of the TC2 First Working IFIP Conf. on Software Architecture (WICSA1)*, pages 35–50. Kluwer, B.V., 1999.

[MLH^{+}00] M. Miguel, T. Lambolais, M. Hannouz, S. Betgè-Brezetz, and S. Piekarec. Uml extensions for the specification and evaluation of latency constraints in architectural models. WOSP2000, pages 83–88, 2000.

[Mol82] M. K. Molloy. Performance analysis using stochastic petri nets. *IEEE Transactions on Software Engineering*, 31:739–743, 1982.

[Mos99] L. Moss. Coalgebraic logic. *Ann. Pure & Appl. Logic*, 1999.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100(1):1–77, 1992.

[MS96] F. Moller and P. Stevens. The edinburgh concurrency workbench (version 7). User's manual, LFCS, Edinburgh University, 1996.

[NA03] O. Nierstrasz and F. Achermann. A calculus for modeling software components. In F. S. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. First International Symposium on Formal Methods for Components and Objects (FMCO'02)*, pages 339–360. Springer Lect. Notes Comp. Sci. (2852), 2003.

[ND95] O. Nierstrasz and L. Dami. Component-oriented software technology. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice-Hall International, 1995.

[NTdMS91] Oscar Nierstrasz, Dennis Tsichritzis, Vicki de Mey, and Marc Stadelmann. Objects + scripts = applications. In *Proceedings, Esprit 1991*

Conference, pages 534–552, Dordrecht, NL, 1991. Kluwer Academic Publishers.

[Oqu04] F. Oquendo. π-adl: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *SIGSOFT Softw. Eng. Notes*, 29(3):1–14, 2004.

[OR04] J. N. Oliveira and C. J. Rodrigues. Transposing relations: From *Maybe* functions to hash tables. In D. Kozen, editor, *7th International Conference on Mathematics of Program Construction*, pages 334–356. Springer Lect. Notes Comp. Sci. (3125), July 2004.

[PA98] G. Papadopoulos and F. Arbab. Coordination models and languages. In *Advances in Computers — The Engineering of Large Systems*, volume 46, pages 329–400. 1998.

[Par81] D. Park. Concurrency and automata on infinite sequences. pages 561–572. Springer Lect. Notes Comp. Sci. (104), 1981.

[Pat95] Fabio D. Paternò. *A Method for Formal Specification and Verification of Interactive Systems*. PhD thesis, Department of Computer Science, University of York, 1995. Available as Technical Report YCST 96/03.

[Pet00] D: Petriu. Deriving performance models from uml models by graph transformations. Tutorial in WOSP2000, 2000.

[PK99] R. Pooley P. King. Derivation of petri net performance models from uml specifications of communication software. Proc. of XV UK Performance Engineering Workshop, 1999.

[Poo99] R. Pooley. Using uml to derive stochastic process algebra models. Proc. of XV UK Performance Engineering Workshop, 1999.

[PS01] A. Ponse and Scott A. Smolka, editors. *Handbook of Process Algebra*. Elsevier Science Inc., New York, NY, USA, 2001.

[PV02] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11):1056–1076, 2002.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[PW99] D. Petriu and X. Wang. From uml descriptions of high-level software architectures to lqn performance models. *Proc. of AGTIVE'99*, 1779:47–62, 1999.

[RBB06] P. Ribeiro, M. A. Barbosa, and L. S. Barbosa. Generic process algebra: A programming challenge. *Journal of Universal Computer Science*, 12(7):922–937, 2006.

[RBW07] P. Ribeiro, P. Barbosa, and S. Wang. An exercise on transition systems. *Electr. Notes Theor. Comput. Sci.*, 207:89–106, 2007.

[Rei81] H. Reichel. Behavioural equivalence — a unifying concept for initial and final specifications. In *Third Hungarian Computer Science Conference*. Akademiai Kiado, Budapest, 1981.

[Rey83] J. C. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing 83*, pages 513–523, 1983.

[RFM91] Mark Ryan, José Fiadeiro, and Tom Maibaum. Sharing actions and attributes in Modal Action Logic. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, pages 569–593. Springer Lect. Notes Comp. Sci. (526), 1991.

[RP99] P. King R. Pooley. The unified modeling language and performance engineering. In *Proc. of IEE Software*, 1999.

[RS95] J. A. Rolia and K. C. Sevcik. The method of layers. *IEEE Transactions on Software Engineering*, 21(8):682–688, 1995.

[RT94] J. Rutten and D. Turi. Initial algebra and final co-algebra semantics for concurrency. In *Proc. REX School: A Decade of Concurrency*, pages 530–582. Springer Lect. Notes Comp. Sci. (803), 1994.

[Rut00] J. Rutten. Universal coalgebra: A theory of systems. *Theor. Comp. Sci.*, 249(1):3–80, 2000. (Revised version of CWI Techn. Rep. CS-R9652, 1996).

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[Sim90] A. Simonič. Grupe operatorjev s pozitivnim spektrom. Master's thesis, Univerza v Ljubljani, FNT, Oddelek za Matematiko, 1990.

[SN99] J.-G. Schneider and O. Nierstrasz. Components, scripts, glue. In L. Barroca, J. Hall, and P. Hall, editors, *Software Architectures - Advances and Applications*, pages 13–25. Springer-Verlag, 1999.

[SW97] C. U. Smith and L. G. Williams. Performance engineering evaluation of object oriented systems with spe•ed. 1245, 1997.

[Szy98] C. Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[TP97] D. Turi and G.D. Plotkin. Towards a mathematical operational semantics. In *Proc. 12^{th} LICS Conf.*, pages 280–291. IEEE, Computer Society Press, 1997.

[Tur95] David Turner. *The Polymorphic π-calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995.

[Tur96] D. Turi. *Functorial Operational Semantics and its Denotational Dual*. PhD thesis, Free University of Amsterdam, June 1996.

[VC00] R. Mirandola V. Cortellessa. Deriving a queueing network based performance model from uml diagrams. WOSP2000, pages 58–70, 2000.

[VU97] V. Vene and T. Uustalu. Functional programming with apomorphisms (corecursion). In *Proc. 9th Nordic Workshop on Programming Theory*, 1997.

[W2C05] W2C. Web Services Choreography Description Language (version 1.0). `www.w3.org/TR/ws-cdl-10/`, 2005.

[W2C07] W2C. Web Services Description Language (version 2.0). `www.w3.org/TR/wsdl20/`, 2007.

[WF98] M. Wermelinger and J. Fiadeiro. Connectors for mobile programs. *IEEE Trans. on Software Eng.*, 24(5):331–341, 1998.

[WN95] G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. M. Gabbay, and T. S. E. Gabbay, editors, *Handbook of Logic in Computer Science (vol. 4)*, pages 1–148. Oxford Science Publications, 1995.

[WNPM95] C. Woodside, J. Neilson, S. Petriu, and S. Mjumdar. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Transaction on Computer*, 44:20–34, 1995.

[WS-07] Business Process Execution Language for Web Services (version 1.1). www.ibm.com/developerworks/library/specification/ws-bpel/, 2007.

[WW99] P. Wadler and K. Weihe. Component-based programming under different paradigms. Technical report, Dagstuhl Seminar 99081, February 1999.

[ZXCH07] Qui Zongyan, Zhao Xiangpeng, Cai Chao, and Yang Hongli. Towards the theoretical foundation of choreography. In P. Patel-Schneider and P. Shenoy, editors, *Proceedings of the 16th Int Conf. on World Wide Web*, pages 973–982. ACM, 2007.