Components, Scripts, and Glue: A conceptual framework for software composition

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Jean-Guy Schneider

von Pierterlen, BE

Leiter der Arbeit: Prof. Dr. O. Nierstrasz,

Institut für Informatik und angewandte Mathematik

Components, Scripts, and Glue: A conceptual framework for software composition

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Jean-Guy Schneider

von Pierterlen, BE

Leiter der Arbeit: Prof. Dr. O. Nierstrasz,
Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Der Dekan:

Bern, den 28. Oktober 1999 — Prof. Dr. A. Pfiffner

Abstract

The last decade has shown that object-oriented technology alone is not enough to cope with the rapidly changing requirements of present-day applications. Typically, object-oriented methods do not lead to designs that make a clear separation between computational and compositional aspects. Component-based systems, on the other hand, achieve flexibility by clearly separating the stable parts of systems (i.e. the *components*) from the specification of their composition. Components are black-box entities that encapsulate services behind well-defined interfaces. The essential point is that components are not used in isolation, but according to a *software architecture* which determines the interfaces that components may have and the rules governing their composition. A component, therefore, cannot be separated from a *component framework*.

Naturally, it is not enough to have components and frameworks, but one needs a way to plug components together. However, one of the main problems with existing languages and systems is that there is no generally accepted definition of how components can be composed. In this thesis, we argue that the flexibility and adaptability needed for component-based applications to cope with changing requirements can be substantially enhanced if we do not only think in terms of *components*, but also in terms of *architectures*, *scripts*, and *glue*. Therefore, we present a *conceptual framework for component-based software development* incorporating the notions of components and frameworks, software architectures, glue, as well as scripting and coordination, which allows for an algebraic view of software composition.

Furthermore, we define the FORM calculus, an offspring of the asynchronous π-calculus, as a formal foundation for a composition language that makes the ideas of the conceptual framework concrete. The FORM calculus replaces the tuple communication of the π-calculus by the communication of *forms* (or extensible records). This approach overcomes the problem of position-dependent arguments, since the contents of communications are now independent of positions and, therefore, makes it easier to define flexible and extensible abstractions.

We use the FORM calculus to define a (meta-level) framework for concurrent, object-oriented programming and show that common object-oriented programming abstractions such as instance variables and methods, different method dispatch strategies as well as synchronization are most easily modelled when *class metaobjects* are explicitly reified as first-class entities and when a *compositional view* of object-oriented abstractions is adopted. Finally, we show that both, polymorphic form extension and restriction are the basic composition mechanisms for forms and illustrate that they are the key concepts for defining extensible and adaptable, hence reusable higher-level compositional abstractions.

Acknowledgements

This thesis could not have been written without the support of many people, and I would like to thank all those who have helped me along the way, knowing that I will never be able to truly express my appreciation.

First of all, I would like to thank my supervisor Prof. Oscar Nierstrasz for giving me the opportunity to work on this topic. He has given me the encouragement, the freedom, and the responsibility I needed to complete this work. He has always been available for discussions when I needed them, and I feel privileged to have worked under his supervision.

A big thank you goes to all members of the "formal gang", especially to my office mate Markus Lumpe, who helped me many times during the last few years, and motivated me to continue in hard times.

I would like to thank all the other (past and present) members of the Software Composition Group who made working in this group enjoyable.

Special thanks go to Franz Achermann, Serge Demeyer, Stéphane Ducasse, Edgar Lederer, and Markus Lumpe who reviewed early drafts of this thesis. Their comments greatly helped to improve my thesis.

I would like to thank Dave Beazley, Douglas Cunningham, Cameron Larid, Doug Lea, Mark Lutz, Guido van Rossum, Aaron Watters, Brent Welch, and all the tutorial attendees of OOPSLA 1998 and ECOOP 1999 who helped me to get further insights in the area of scripting and to clarify my ideas.

I would like to thank Benjamin Pierce and David Turner for giving me insight in both the π-calculus and the PICT programming language and commenting on early modelling stages.

Working in an UNIX environment is never possible without the helping hands of the local system administration. Therefore, many thanks to Heike Horn and Jürg Stiefenhofer.

Thanks to all my colleagues and friends from Tennis who cheered me up when necessary and pulled me back to earth when I was getting lost in "higher spheres" of mathematics.

A very special thank you goes to my family. I would not have been able to do this work without their help, and their support is greatly appreciated. I should not forget to mention my cat Niki who kept me company during many hours working at home; unfortunately, he has not been able to assist me during the last few weeks of my work...

I dedicate this thesis to my mother, Rosmarie Schneider. Her love and encouragement has been a constant support for me. Thank you very much.

Contents

I Introduction 1

1 Introduction, Background 3

1.1 Background 3
1.2 Contributions 5
1.3 Outline 6
1.4 Disclaimer 6

2 Component-based software development 7

2.1 Motivation 8
2.2 Components and frameworks 9
2.3 State-of-the-art in component technology 12
2.4 A conceptual framework for software composition 14
2.5 Requirements for a composition language 16

II A conceptual framework for software composition 19

3 Software architectures 21

3.1 What is an architecture? 21
3.2 Roots of software architecture 24
3.3 Definitions of software architecture 26
3.4 Architectural styles and patterns 30
3.4.1 What are architectural styles? 30
3.4.2 Classifying architectural styles 31
3.4.3 Architectural patterns 32
3.5 Architectural description languages 33
3.6 Influence of software architectures on software engineering 34
3.7 Related aspects 37
3.8 Summary 39

4 Scripting 40

4.1 What is scripting? 40
4.2 Dimensions of scripting languages 46
4.2.1 Essential features 47

4.2.2 Characterizing features . . . 49
4.2.3 Related aspects . . . 51
4.3 Systems and languages . . . 52
4.3.1 Bourne Shell . . . 52
4.3.2 Tcl . . . 54
4.3.3 Perl . . . 56
4.3.4 Python . . . 57
4.3.5 AppleScript . . . 59
4.3.6 Manifold . . . 62
4.3.7 Summary of concepts . . . 65
4.4 Dynamic aspects of scripting languages . . . 66
4.4.1 Scoping rules, namespaces . . . 66
4.4.2 Eval . . . 67
4.5 Summary . . . 72

5 Glue **73**
5.1 What is glue? . . . 73
5.2 Catalogue of glue problems . . . 76
5.3 Glue technology . . . 80
5.3.1 Ad-hoc techniques . . . 81
5.3.2 Wrapping techniques . . . 81
5.3.3 Intermediate forms . . . 83
5.3.4 Patterns . . . 83
5.3.5 Reflective approaches . . . 84
5.4 Discussion . . . 85

6 Unification of concepts **87**
6.1 Concepts in practice . . . 87
6.2 Discussion . . . 91

III Towards a composition language **93**

7 Towards a composition calculus **96**
7.1 The polyadic mini π-calculus . . . 97
7.2 The $\pi\mathcal{L}$-calculus . . . 99
7.3 The FORM calculus . . . 101
7.3.1 Names and forms . . . 102
7.3.2 The language . . . 108
7.3.3 Encoding of booleans and choice . . . 108
7.3.4 Name binding . . . 110
7.3.5 Form substitution . . . 111
7.3.6 α-substitution . . . 112
7.3.7 Operational semantics . . . 113

7.3.8 Observable equivalence 118
7.3.9 α-conversion 120
7.3.10 Encoding the mini π-calculus in the FORM calculus 122
7.3.11 Encoding the FORM calculus in the mini π-calculus 123
7.4 Comparison of the FORM calculus with the $\pi\mathcal{L}$-calculus 131

8 Modelling objects in the FORM calculus 133
8.1 Basic object model in the FORM calculus 134
8.1.1 The Pierce/Turner basic object model 134
8.1.2 Modelling requirements 137
8.1.3 Class and object features 138
8.1.4 Dynamic binding and encapsulation 140
8.1.5 Single inheritance 142
8.1.6 Observations 146
8.2 Synchronization 147
8.2.1 Generic Synchronization Policies 148
8.2.2 Modelling GSPs in the FORM calculus 150
8.2.3 Observations 153
8.3 Class abstractions 154
8.3.1 From pre-methods to generators 154
8.3.2 Smalltalk-like behaviour 155
8.3.3 Static and dynamic binding 159
8.3.4 Beta-style inheritance 163
8.3.5 Multiple inheritance 165
8.4 Mixins 167
8.4.1 Basic mixin abstraction 168
8.4.2 Mixin composition 169
8.4.3 Examples revisited 172
8.4.4 Singleton mixin 173
8.5 Applying form restriction 174
8.6 A meta-level framework 177
8.6.1 The MetaModel abstraction 178
8.6.2 Model abstractions for classes 179
8.6.3 Model abstractions for mixins and encapsulation 181
8.7 Form introspection 183
8.8 Comparison with other meta-level approaches 185
8.8.1 Structure of classes 186
8.8.2 Structure of instances 188
8.8.3 Operations 188
8.8.4 Inheritance 189
8.8.5 Method dispatch 190
8.8.6 Adaptability and extensibility 192
8.9 Related work 193

8.10 Summary . . . 195

9 Compositional abstractions 198
9.1 A framework for stream and filter composition . . . 198
9.2 Dispatch-based approaches . . . 200
9.2.1 Single dispatch . . . 200
9.2.2 Double dispatch . . . 201
9.2.3 Multiple dispatch, method overloading . . . 204
9.3 Operator-based approaches . . . 204
9.4 Discussion . . . 207

10 Summary of observations 210

IV Conclusions 213

11 Conclusions and future work 215
11.1 Conclusions . . . 215
11.2 Future work . . . 217

V Appendices 221

A Source code of UniBE phone book application 223
A.1 Stream framework . . . 223
A.2 String dictionaries . . . 227
A.3 HTML parsing filter . . . 227

B Congruence of weak bisimulation 229

C PICCOLA(F) **language definition 234**

D Filter algebra 236

Glossary 239

Bibliography 241

Part I

Introduction

Chapter 1

Introduction, Background

This thesis summarizes the state-of-the-art in software architectures, scripting languages, and glue technology, and discusses their influence on component-based software development. The discussion motivates requirements for a general-purpose composition language that integrates the concepts of component frameworks, software architectures, scripting, glue, and coordination. These requirements lead to the definition of the FORM calculus – a conservative variant of the π-calculus where tuple-communication is replaced with communication of extensible records (or *forms*) – as a formal semantic foundation for a composition language. The suitability of the FORM calculus as a minimal semantic foundation is validated by using it for modelling compositional abstractions as well as for modelling a meta-level framework incorporating common features of concurrent, object-oriented programming languages.

1.1 Background

Now, more than ever, it is not enough for applications to fulfill only their functional requirements. Modern applications must be *flexible*, or "open" in a variety of ways in order to cope with the advances in computer hardware technology and rapidly changing requirements. The Concise Oxford Dictionary defines "flexible" as "that will bend without breaking, pliable, pliant; easily led, manageable; adaptable, versatile; supple, complaisant." Software that "bends without breaking" is *portable* (to different hardware and software platforms), *interoperable* (with other applications), *extensible* (to new functionality), *configurable* (to individual users' or clients' needs), and *maintainable*.

Object-oriented programming languages and design techniques go a long way in offering the required flexibility and extensibility, but current practice shows that the technology is often applied in a way that hinders the development of open systems [Ude94]. The kinds of flexibility required by open systems are presently best supported by component-oriented software technology: components, by means of abstraction, support portability, interoperability, and maintainability. Extensibility and configurability are supported by different forms of binding technology: application parts, or even whole applications, can be created by composing software components. This ensures that applications stay flexible by allowing components to be replaced or reconfigured, possibly at run-time.

Today, there is considerable experience in component technology, and a lot of resources are spent in defining components and component models (such as COM [Rog97], CORBA [OMG96], or Enterprise JavaBeans [Sun99] to name just a few). However, much less effort is spent in investigating appropriate composition or *wiring* technology.

Specialized scripting languages and 4GLs each provide different forms of wiring: it can be either highly structured, reflecting a particular architectural style of application composition, or it may be largely unstructured. Using a structured wiring technology generally constrains flexibility by limiting the ways in which components may be configured, but supports application maintainability and comprehension by making application architectures explicit [MDK92]. On the other hand, an unstructured binding technology is ultimately more flexible, but at the cost of maintainability: Tcl, for example, is a simple scripting language that is good for combining arbitrary C functions for simple configuration tasks, but it does not support programming in the large, and, if abused, can lead to large, unmaintainable scripts [Ous94].

In an ideal component world, there are components available for any task applications have to perform and these components can be simply plugged together. However, it is a fact that a user is often constrained to work with (legacy) components that are not plug compatible with the components he has to work with. In such a situation, glue code is necessary in order to overcome compositional mismatches and to make components which otherwise cannot be plugged together composable. Glue code may be ad hoc, written to adapt a single component, or it may consist of generic abstractions which can be reused in similar settings.

Although object-oriented languages are well-suited for implementing software components, they fail to shine in the construction of component-based applications, largely because object-oriented design tends to obscure a component-based architecture. Using a *composition language* that allows us to express applications as compositions in terms of components, scripts, and glue, would be a major step in overcoming these problems.

However at present, there does not exist a general-purpose composition language that i) supports application configuration through a structured, but nevertheless flexible wiring technology and ii) enforces a clear separation between computational elements (i.e. components) and their relationships. In addition, most available systems mainly focus on special application domains and offer only rudimentary or no support for the integration of components not built within the system. The reason for this situation is not only the lack of well-defined (or standardized) component interfaces, but also the ad-hoc way the semantics of the underlying language models are defined.

In order to solve the problems of present-day component technology, we argue that it is necessary to define a composition language based on an appropriate semantic foundation. In particular, if we can understand all aspects of software components and their composition in terms of a small set of primitives, then we have a better hope of being able to cleanly integrate all required features for software composition in one unifying concept.

1.2 Contributions

In this dissertation, I illustrate that

> Making a clear separation between computational elements and their relationships enhances the flexibility, maintainability, and robustness of software systems. This concept can be most naturally expressed in terms of a formal foundation that includes asymmetric record concatenation and restriction.

More specifically, the contributions of the thesis can be summarized as follows:

– We summarize the state-of-the-art in component technology, analyze advantages and drawbacks of existing approaches, and identify the need to make a clear separation between computational elements and their relationships. Based on this observation, we propose a conceptual framework for software composition, incorporating the concepts of component frameworks, software architectures, scripting, glue, and coordination, and define a set of requirements for a general-purpose composition language.

– We clarify the terms software architecture, architectural style as well as glue abstraction and analyze their importance in the context of component-based software development. In particular, we illustrate the relationship between architectural styles and component frameworks from an algebraic point of view and discuss the influence of software architectures on reuse. Furthermore, we define a catalogue of glue problems and summarize known glue technology.

– We analyze the concept of scripting as the key mechanism for wiring components together. We give a general introduction to scripting, discuss the main properties and abstractions of scripting languages, and identify a list of essential and characterizing features. Furthermore, we compare selected scripting languages in order to illustrate important concepts of scripting.

– Based on the ideas of Milner [MPW92], Dami [Dam94], and Lumpe [Lum99], we define the FORM calculus, a variant of the π-calculus where the communication of tuples is replaced with the communication of labelled parameters, or *forms*. In fact, in the FORM calculus, parameters are identified by names rather than positions which provides a higher degree of flexibility, extensibility, and robustness for defining compositional abstractions. We give a basic theory for the FORM calculus, illustrate the main differences in comparison with the π-calculus, and define an asynchronous bisimulation relation on FORM calculus agents.

– We use the FORM calculus to define a (meta-level) framework for concurrent, object-oriented programming and show that common abstractions such as instance variables and methods, different method dispatch strategies as well synchronization are most easily modelled when *class metaobjects* are explicitly reified as first-class entities. We illustrate that various concepts which are typically merged (or confused) in object-oriented programming languages can be expressed in a more natural way by making a clear separation between functional elements (i.e. methods) and their compositions (i.e. inheritance). Furthermore, we show that the same concepts can also be applied for modelling mixins, mixin application, and mixin composition.

– Finally, we show that both polymorphic form extension and restriction are the basic composition operations for forms and illustrate that they are the key concepts for defining higher-level compositional abstractions. In fact, by using both concepts, we obtain a powerful mechanism for software composition, since it allows a user to express the composition of the services of a given set of components in a more flexible, extensible, and robust way.

1.3 Outline

This thesis is organized in four parts as follows: in Part I, we give an introduction to our work and discuss the corresponding background. We motivate our work in chapter 2 by summarizing the state-of-the-art in component technology, analyze problems with existing approaches, and define important terms used throughout the rest of this work.

In Part II, we define the notions of software architectures, scripting, and glue (chapters 3 to 5), introduce a conceptual framework for software composition based on these notions, and illustrate how the concepts of components, scripts, and glue are used in practice (chapter 6).

In Part III, we define the FORM calculus, a conservative extension of the π-calculus as a formal foundation of our study (chapter 7). We use the FORM calculus to model objects and concepts found in object-oriented programming languages such as classes, inheritance, and mixins (chapter 8) as well as compositional abstractions (chapter 9). We summarize the main observations of our modellings in chapter 10.

Finally, in Part IV, we summarize the main contributions of our work and conclude with a discussion about related and future work.

1.4 Disclaimer

This thesis is part of ongoing research of the Software Composition Group at the Institute of Computer Science and Applied Mathematics of the University of Berne, which conducts research into tools, techniques, and methods for constructing flexible software systems from components. Several parts of this thesis have been developed in collaboration with other members of the group and, therefore, it is possible that some of the results presented here overlap with results presented by others, in particular with the Ph.D. thesis of Markus Lumpe [Lum99].

Chapter 2

Component-based software development

Present-day applications are increasingly required to be open systems in terms of topology (distributed systems), platform (heterogeneous hardware and software) and evolution (rapidly changing requirements). The last decade has shown that object-oriented technology alone is not enough to cope with the rapidly changing requirements of present-day applications [Ude94]. One of the reasons is that, although object-oriented methods encourage one to develop rich 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 [SN99].

Already in 1969, McIlroy [McI69] has proposed another way to produce software called *component-oriented software construction*. The basic idea of his proposal was that we should not think any more about which mechanism we should use but what mechanism we should build. He viewed components as families of routines which are constructed based on well-defined principles so that these families fit together as building blocks. McIlroy stated that these families constitute components which are *black-box* entities.

Unfortunately, his vision could not be established at this time. There are several reasons for this such as the idea that components should be built system independently or that a component catalogue must be available in order to allow application programmers to choose the right component for a specific problem. In the last few years, even not directly related to McIlroy's work, *component-based software development* has become very popular. Component-based systems achieve flexibility by clearly separating the stable parts of the system (i.e. the 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. Components are not used in isolation, but according to a software architecture that determines the interfaces that components may have and the rules governing their composition.

In this chapter, we motivate our work by summarizing the state-of-the-art in component technology and analyzing problems with existing approaches. We define several important terms which we will use throughout this work and introduce a conceptual framework for software composition as an approach to overcome the problems of current component technology. We conclude this chapter with a discussion of requirements for a general-purpose composition language based on the conceptual framework.

2.1 Motivation

In order to cope with the advances in computer hardware technology and rapidly changing requirements, there has been a continuing trend in the development of software applications towards so-called *open systems* [Tsi89]. Open systems differ from closed, proprietary systems in the sense that they are not only open in terms of topology (distributed systems) and platform (heterogeneous hardware and software), but particularly in terms of changing requirements: they assume that requirements evolve rapidly and are neither closed nor stable. In fact, to address evolution, each individual application should be viewed as an instance of a *generic class* of applications, and an essential point is that open systems define a *generic* (hence reusable) *architecture* for a family of related applications. Furthermore, an individual application may either be considered as an instance of a family of applications or a snapshot in time of an evolving application [ND95]. By viewing open systems as compositions of reusable and configurable software components, we expect to better cope with the requirements of present day applications in general and rapidly evolving requirements in particular.

Object-oriented programming languages and analysis and design methods provide a well-suited tool-box for component-based application development, but current practice shows that the technology is often applied in a way that hinders the development of open systems.

Object-oriented analysis and design methods are domain-driven, which usually leads to designs based on domain objects. Most of these methods make the assumption that an application is being built from scratch, and they incorporate reuse of existing architectures and components too late in the development process (if at all) [Ree96].

In order to successfully plug components together, it is necessary that i) the interface of each component matches the expectations of the other components and that ii) the "contracts" between the components are well-defined. Therefore, component-based application development depends on adherence to restricted, plug-compatible interfaces and standard interaction protocols. However, the result of an object-oriented analysis and design method generally is a design with rich object interfaces and non-standard interaction protocols.

Object-oriented programming languages have been very successful for implementing and packaging components, but they only offer limited abstractions for flexibly connecting components and explicitly representing architectures in applications. Given the source code of an object-oriented application, one can more easily identify the components, but it can be notoriously difficult to tell how the system is composed. The reason is that object-oriented source code exposes *class hierarchies*, not *object interactions*. In addition, the way objects are interconnected is typically distributed amongst the objects themselves, which hinders a clean separation between computational and compositional aspects.

Although object-oriented applications can often be adapted to additional requirements with a minimal amount of new code, it can require a great deal of detailed study in order to find out where exactly the extension is needed. Unfortunately, object-oriented frameworks do not make their generic architecture explicit, which results in a steep learning curve before a framework can be successfully used. Since object-oriented frameworks focus on subclassing of framework classes (also known as *white-box reuse*), a detailed understanding of the generic architecture is needed in order to prevent contracts between two classes from being violated. In addition, changing framework classes often implies extensive modifications of application-specific code.

Recently, *visual programming* has attracted widespread attention by promising to make programming much easier [Mey94, AEW96]. Visual application builders go a step further than object-oriented frameworks, since they already incorporate important ideas and concepts needed for component-based application development (e.g. higher-level abstractions for composing components). Furthermore, visual builders generally focus on a specific application domain (e.g. Delphi focuses on database applications for the Windows platform [Bor95]) and consist of an integrated development environment (IDE) with various tools that can be used to quickly create graphical user interfaces and for applying event-driven programming techniques. Visual application building, however, rests on a programming model of plug-and-play of prefabricated components. The number (and type) of components that are combined is usually quite small, and the mutual dependencies between these components take simple (standard) forms.

Despite their usefulness and contribution to increased software productivity and reuse, components of visual programming systems share one common handicap: they need their specific support environment and cannot be easily combined across different systems. Due to their restriction to specific application domains, visual application builders are not flexible enough for general-purpose component-based development and generally lack a well-understood formal foundation.

2.2 Components and frameworks

We have previously used the terms component and framework without giving a precise definition. However, for the rest of this work it is essential that we have a clear understanding what a component is and how frameworks look like. From our point of view, both terms cannot be defined in isolation and are closely related, which motives the following definition:

> A *software component* is a composable element of a component framework [LSNA97].

Although this definition seems to be circular, it captures the essential properties of components: components are designed to be plugged together with other components. A single component that does not belong to a component framework is a contradiction in terms. Furthermore, a component can in general not function outside a well-defined framework.

As an example, consider a stereo system which consists of an amplifier, CD-player, tuner, tape deck, and other components. Each of these stereo components is built up from smaller components (e.g. circuits), which again use even smaller components (e.g. transistors). Although in principle each of these components would function by itself, their real value lies in the way they are designed to be plugged together.

> A software component is a *static abstraction with plugs* [ND95].

A component is a black-box abstraction that hides implementation details and must be instantiated in order to be used (this is imposed by the characterization as "static abstraction"). In object-oriented programming languages, we distinguish between *classes* and instances of classes (i.e. *objects*). Unfortunately, we do not have such a convenient way to distinguish between components and their instances, which is a frequent source of confusion. Talking about components, we often mean their instances. The different roles of components and their instances are hidden by the way component-based software is developed: composition is usually done within a composition environment where the difference between components and their instances vanishes. An application programmer, even at design time, always works with instances.

A component has *plugs*, which are not only used to *provide* services, but also to *require* them.[1] A true black-box component advertises all of its features and dependencies by means of public plugs; there are no hidden dependencies. Plugs are the main prerequisite for composition: connections between components are established by connecting required services with appropriate provided services. A plug is a pluggable interface, but the kind of interface it exactly is and how these interfaces are plugged together, may depend from one component framework to another.

> A *software component* is a unit of independent deployment, a unit of third-party composition, and has no persistent state [Szy98].

This definition has several implications. First, for a component to be independently deployable, it needs to be well separated from its environment and from other components. Therefore, a component *encapsulates* its constituent features and can never be deployed only partially. Second, for a component to be composable with other components, it needs to be sufficiently *self-contained* and must explicitly specify the services it provides, but also the services it requires. Hence, a component needs to encapsulate its implementation and interact with its environment through well-defined interfaces. Finally, for a component not to have any persistent state, it is required that it cannot be distinguished from copies of its own.

[1] Some authors denote provided services as *sockets* and required services as *plugs*. However, in order to treat both concepts uniformly, we will not use separate terms throughout the rest of this work.

As an example of a system which conforms to the definitions given above, consider the Bourne Shell [Bou78]. It offers a simple component model based on *commands* where each command has the ability to read from the standard input stream and produce output onto the standard output and/or error streams. In our terminology, the standard input stream is a required service of a command whereas both the standard output and error streams are provided services. Commands can be composed using the pipe operator '|' which connects the plugs of a pair of provided and required services (i.e. standard output stream of one component with the standard input stream of another component). A composition of commands is again a command which can be used in other so-called Shell scripts. For a detailed discussion about the Bourne Shell, refer to section 4.3.

As we will further discuss in section 3.1, the components and the composition mechanisms of the Bourne Shell restrict the kinds of applications which can be implemented. For example, it is not possible to define *feedback loops* between commands. In addition, all Shell scripts conform to the same architectural structure (i.e. a *pipe and filter* architectural style). These observations motivate the following definition:

> A *component framework* is a collection of software components with a software architecture that determines the interfaces that components may have and the rules governing their composition [SG96].

This definition closes the loop. The essential point is that components are not used in isolation, but in context with a software architecture that determines how components are plugged together. In contrast to an object-oriented framework, where an application is generally built by subclassing framework classes that respond to specific application requirements (also known as *hot spots* [Pre95]), a component framework primarily focuses on object and class composition (i.e. *black-box* reuse). The reader should note that the architecture of a component framework (i.e. the kinds of components and the rules of their composition) may vary quite dramatically from one framework to another.

Another definition of the term component framework is given by Szyperski [Szy98]. He describes a component framework as a set of interfaces and rules of interactions that govern how components plugged into the framework may interact. He also points out that an overgeneralization of that scheme has to be avoided in order to keep actual use of frameworks practicable.

Naturally, it is not enough to have components and frameworks, but for building real applications, one needs a way to wire components together (i.e. to express *compositions*). The idea behind component-based development is that an application developer only has to write a small amount of *wiring code* in order to establish connections between components. The wiring technology, or *scripting*, can take take various forms, depending on the nature and granularity of the components, the nature and problem domain of the framework, and the composition model. Composition may occur at compile-time, link-time, or run-time, and may be very rigid and static (like the syntactic expansion that occurs when C++ templates are composed [MS96]), or very flexible and dynamic (like that supported by Tcl or other scripting languages [Ous98]).

In an ideal world, there are components available for any task an application has to perform and these components can be simply plugged together. However, it is sometimes necessary to reuse a component in a different environment than the one it has been designed for and that this environment does not match the assumptions the component makes about the structure of the system to be part of. In such a situation, *glue* code is needed to overcome the mismatched assumptions and to adapt components in order to be composable.

Even though two components may fit together perfectly by means of their interfaces, they may require different run-time or operating systems. Unless they can communicate across different processes, this is a hindrance for composing such components. In order to make these (often implicit) assumptions of components explicit, Sametinger defines the notion of a *component platform*, which indicates any soft- or hardware a component is built upon [Sam97]. Typical examples of component platforms are operating systems, run-time systems, window systems, compilers, libraries, and network connections. In general, platforms are collections of homogeneous components.

If components are classified by their platforms, it is possible to determine the reusability of a component in a certain context. Based on the role a platform plays for a component, Sametinger distinguishes between *execution platforms* and *composition platforms*. Execution platforms are necessary in order to correctly execute a component and determine the environment in which a component (or a complete running system) can be executed. Typically, a component requires a certain operating system for execution, but it may also require a specific hardware and/or programming system (e.g. an interpreter). Composition platforms are necessary for components that do not run on their own, but rather have to be integrated with other components to form an executable application. Compilers and linkers are the most prominent form of this category.

2.3 State-of-the-art in component technology

There is a large variety of programming languages and systems emerging from both industry or academia integrating the notion of components. Each of these systems defines its own component model and incorporates different composition mechanisms. In the following, we briefly discuss important properties of selected languages and systems. A detailed comparison of component models and systems is beyond the scope of this work; refer to [Sam97, Szy98, Lum99] for further details.

The composition mechanism of Visual Basic is based on a scripting approach [Mic97]. Visual Basic components (in the following also called Visual Basic controls) are plugged together with the dynamically typed language Visual Basic, which is an object-oriented extension of BASIC. For components, there exists a special type called `VARIANT` which is implicitly used by the system. This type defines the general interface of all components in the system. Unfortunately, this type does not allow programmers to use arbitrary (user defined) types in component interfaces; only the types which are supported by the `VARIANT` type implementation can be used. For an application programmer this is generally not a problem because type information is never exposed to the programmer himself; only the provider of a Visual Basic control has to keep in mind that this restriction exists, especially when languages like C or C++ are used to develop Visual Basic controls.

Delphi [Bor95] and JavaBeans [Sun97b] use a general-purpose programming language for composition. These languages (i.e. Object Pascal for Delphi and Java for JavaBeans) are strongly typed, which limits the flexibility of composition. It is also not easy to integrate new component models because they have to fulfill certain constraints according to the type system of the programming language. In both systems, plugging components together implies that a new class[2] has to be defined where the composed components are specified as member variables. Furthermore, both systems use event handlers for composition: when an event is raised, a user-defined handler is called which implements the wiring between the components.

For D-Active-COM-X-++ the situation is different. The *Component Object Model* (COM) defines a language independent component model [Rog97]. For this reason, no composition language is defined for COM, and COM components (also known as *ActiveX controls*) are usually developed in either C or C++.[3] COM defines a so-called *binary standard* for ActiveX controls which must be fulfilled by every component. The memory layout for the interface of such a component is the same as the memory layout that is generated for abstract base classes by the Microsoft C++ compiler. This makes it very easy to use C++ to develop ActiveX components. However, it is not required to use an object-oriented programming language to develop COM components, because the COM specification is language neutral.

The Object Management Group has proposed the *Common Object Request Broker Architecture* (CORBA) as an answer to the need for interoperability among a number of available hardware and software products [OMG96]. CORBA allows an application to communicate with another application without knowing where it is located or who has designed it. CORBA defines an Interface Definition Language (IDL) and an Application Programming Interface (API) that enables a programmer to develop client/server object interactions within a specific implementation of an Object Request Broker (ORB). The ORB provides the necessary services to find an object's implementation, to pass parameters, to invoke methods, and to return results. Everything is done transparently for an application programmer. CORBA enables a priori the development of distributed applications which is seen as one of the main contributions of CORBA in comparison with other systems.

CORBA allows the integration of existing components by using *interface wrappers*. In this approach, an application programmer has to define a corresponding interface using the interface description language IDL. An application programmer has to write the code which translates between the CORBA interface (in IDL) and the component interface. The way new components can be integrated in an application is similar as defining a new CORBA component from scratch.

[2]The abstraction *class* is used to describe objects as well as components.

[3]Since version 5.0, ActiveX controls can be defined within Visual Basic.

It is important to note that the IDL is not sufficient for composing components. The IDL can only be used to *describe* an interface, but in order to plug components together, a real programming language is needed. In many cases this is either C, C++, Smalltalk, or Java. Therefore, CORBA on its own cannot be seen as a composition system or language. It has to be considered as *middleware* which provides the necessary services to develop (distributed) applications using CORBA components [OHE95].

A slightly different approach is used for RAPIDE, an event-based, concurrent, object-oriented language specifically designed for prototyping system architectures [LV95]. RAPIDE is intended for modelling the architectures of concurrent and distributed systems, both hardware and software. In order to represent the behaviour of distributed systems in as much detail as possible, RAPIDE is designed to make the greatest possible use of event-based modelling by producing causal event simulations. When a RAPIDE model is executed, it produces a simulation that shows not only the events that make up the model's behaviour and their time-stamps, but also which events caused other events, and which events happened independently.

The main purpose of RAPIDE is to specify and verify properties of specific architectures. However, it is not intended as a language for general-purpose composition since it only focuses on applications with event-based architectures. In addition, components have to be both specified and implemented in the RAPIDE environment, and there does not exist an interface to integrate components written in other programming languages.

The last system which we briefly discuss is Python, an interpreted, object-oriented scripting language with an extensible and embeddable architecture [vR96]. Python has a number of powerful constructs which make it almost ideal both as a scripting system and as a rapid application development tool. From our point of view, a major strength of Python is its extensibility, which is heavily due to the unifying concept of everything being an object, objects being first-class values, and the possibility to use keyword-based parameters. Many extensions have been released for Python like a Tk interface or an OLE interface, and especially the OLE extension has exposed very impressively the power of the extensible architecture. A major drawback of Python is the lack of platform independent concurrency: available extensions depend on the underlying operating system and are in general not portable. For a detailed discussion about the main features of Python, refer to section 4.3.

2.4 A conceptual framework for software composition

As previously discussed in this chapter, complex software systems are increasingly required to be open, flexible conglomerations of heterogeneous and distributed software components rather than monolithic heaps of code. This places a strain on old-fashioned software technology and methods that are based on the maxim

Applications = Algorithms + Data.

This equation has some relevance for well-defined and delimited problems and is often applied in imperative programming languages that focus on top-down decomposition.

Object-oriented programming languages go a step further, are well-suited for encapsulating state and behaviour, and are based on the maxim

Applications = Objects + Messages.

However, as we have discussed in section 2.1, current practice shows that object-oriented technology is often applied in a way that hinders the development of flexible and adaptable, hence open systems. For component-based software engineering, we need an approach that makes a clear separation between computational and compositional entities. Therefore, a component-oriented style must be based on the maxim

Applications = Components + Scripts.

From our point of view, this maxim is essential for any methodology for component-oriented software development in order to achieve the flexibility and adaptability needed for applications to cope with changing requirements.

If the maxim given above was the only requirement for component-based systems, there should be several languages and systems available, well-suited for a wide range of problem domains and applications. However, as discussed in the previous section, this is not the current status. One of the main problems with existing languages and systems is that there is no generally accepted definition of how components can be composed (i.e. it lacks well-defined concepts for both scripting and glue). This results in the fact that in many applications, traditional (or even ad hoc) techniques are used for implementation. In addition, many of these approaches are applicable only in certain contexts (i.e. specific applications domains or hardware platforms) and do not provide general solutions for a wider range of software engineering problems.

From a different perspective, an application (i.e. a composition of components) can be viewed as a term of a *many-sorted algebra*: components are the operands and the composition mechanisms are the operators. Like in any algebra, operators cannot be applied to any operands, and terms always have a well-defined structure. These observations correspond with the definition of a component framework given in section 2.2.

In addition, an application of an operator to a set of operands (i.e. a composition of components using a connector) is, by definition, again an operand of the algebra. Such an algebraic view of composition is adopted by some component-based systems. As an example, reconsider the Bourne Shell: a composition of two commands using the pipe operator '|' is again a command and, therefore, an operand of the corresponding algebra.

There are component-based systems which apply a more procedural view of composition. As an example, consider Darwin, a programming language for structuring parallel and distributed programs [MDK92]. Like any component-based system, Darwin uses the notion of plugs in order to describe components and their interaction. The main abstractions are i) components and ii) services (i.e. means by which components interoperate). Services are further subdivided into provided and required services and offer a number of service access points (or ports). Component interaction is established by connecting service outputs (provided ports) with service inputs (required ports). In order to establish connections between provided and required ports, Darwin provides a built-in statement

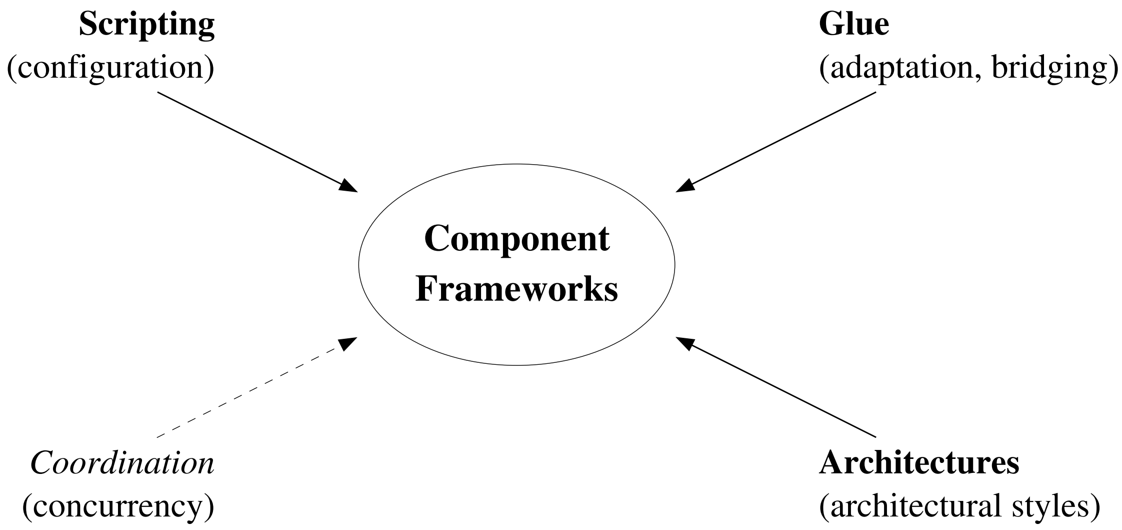


Figure 2.1: Conceptual framework for software composition.

`bind`. In contrast to the Bourne Shell, the statement `bind` simply connects required and provided ports, but it does not yield a new component.

Summarizing the problems with state-of-the-art technology, we argue that the flexibility and adaptability needed for applications to cope with changing requirements can only be achieved if we think *not only* in terms of *components*, but also in terms of *architectures*, *scripts*, and *glue*. Therefore, we claim that a conceptual framework for component-based software development must incorporate the notions of components and frameworks, software architectures, glue, as well as scripting and coordination, and allow for an algebraic view of software composition. For an illustration of this conceptual framework, refer to Figure 2.1.

2.5 Requirements for a composition language

In the previous sections, we have discussed the state-of-the-art in component technology, analyzed problems with existing approaches, and defined a conceptual framework for component-based software development as an approach to overcome these problems. We also identified the need that a programming language for developing component-based systems must allow programmers to make a clear separation between *components* and their *connections*. However, implementing software components and plugging them together are very different activities that may well benefit from different kinds of tools. We expect that traditional programming languages will be best suited for implementing components, whereas something we call a *composition language* may be better for specifying applications as compositions of software components.

In the following, we will discuss a list of requirements for such a composition language, based on previous work in this area (refer to [NM95a, NM95b, NSL96] for details). However, the following list extends previous requirements for a composition language by combining concepts and paradigms of existing languages and systems, and particularly focuses on incorporating the notions of the conceptual framework defined in the previous section.

- **Active Objects:** objects are computational entities that provide services based on an encapsulated state, and a composition language must be able to instantiate and communicate with objects. Objects may be active (concurrent), distributed, mobile, and may live in different environments. In any case, objects can be viewed as a kind of server, or *process*. The process view provides a way to formalize the notion of objects.

- **Components:** components are abstractions over the object space [ND95]. Components may be fine-grained, when used to build individual objects, or coarse-grained, when used to build compositions of objects. They may also be run-time entities, but more generally, components must be constructed (composed) and instantiated before they are part of an application.

- **Connectors:** composition mechanisms and operators (connectors) define how associated components interact with each other. The language model should allow for an algebraic view of composition and encourage a *declarative* style of programming. Furthermore, a composition language must support the specification of new kinds of connectors.

- **Architectures:** a composition language must allow programmers to make the architecture of an application explicit in the corresponding source code. It must offer support for common architectural styles, for combining multiple, heterogeneous architectural descriptions, and be open enough to support new (user-defined) architectures.

- **Glue:** glue abstractions are needed in order to overcome incompatible composition interfaces of components coming from different component frameworks. These glue abstractions have to be *transparent* in the sense that none of the involved components is aware of the glue and *configurable* in order to enhance their reusability.

- **Object Models:** a composition language must be able to mediate between different object and component models. Objects and components that cannot be separated from their individual run-time environments must still be able to communicate. Glue abstractions must achieve the mappings between these models.

- **Reflection:** glue abstractions are often realized by intercepting service invocations between components and by performing some (reflective) transformations on these invocations. Reflection is also important for exercising run-time control on configurations. Metaobjects are active objects that control the creation, instantiation, and composition of other objects, and can be used to realize various forms of reflective behaviour.

- **Foreign Code Concept:** in order to incorporate components not written in the composition language itself, it is necessary that the language offer abstractions to interoperate with components running on different component platforms.

- **Formal Semantics:** a composition language must be based on a formal semantics in order to reason about components and configurations. In particular, a formal foundation helps one to specify different notions of objects and components, to integrate different compositional features such as synchronization or inheritance, and to explore notions of contracts and type compatibility for concurrent systems.

- **High-level Syntax:** the specification of an application as a composition of components must be highly readable and compact. It is therefore important that a user is able to assign a high-level syntax when defining components (as it is possible in languages like Smalltalk).

This list of requirement focuses on technical aspects of a composition language, but does not give any details for specific aspects. In the following chapters, we will elaborate the notions of architectures, scripting, and glue, and also discuss technical requirements for these concepts in further detail; refer to sections 3.5, 4.2, and 5.4, respectively. A detailed discussion about the technical aspects of the other items of the list is beyond the scope of this work. We will comment on some of these aspects in the section about future work (refer to section 11.2).

Part II

A conceptual framework for software composition

Chapter 3

Software architectures

Software architecture is an emerging discipline in the field of software engineering. It did not appear as a new concern for software systems, but it has rather emerged over time as people have searched for new ways to better understand their software and to build large, more complex systems. In this chapter, we will first informally introduce terms and related problems and give a short overview of the roots of software architectures. Second, we define important terms for the field and list a set of requirements how to describe software architectures. Finally, we discuss the influence of software architectures on reuse and conclude with some remarks on related and future work.

3.1 What is an architecture?

Intuitively, people apply the general term architecture to the usage aspects of houses and other buildings they deal with, in terms of the nature of the physical structures and the physical arrangement of the structures in relation to each other [GAACB95]. This first sentence already classifies the most important aspects of architecture: *physical structures* and their *relation to each other*. However, there are other important aspects, and architectures can be found in more areas than just buildings. In this section, we will informally introduce the terms architecture and architectural style using a set of examples from different domains.

A classical introduction into the problems of architecture and architectural styles is offered by modular furniture systems. A piece of such a system is usually delivered as a set of components which have to be assembled before the piece can be used. The components can be divided into *functional* elements (like boards) and *connecting* elements (e.g. screws). The functional elements have appropriate *holes* that can be used by the connecting elements to hold the furniture together. Finally, a *script* shows how (and in which order) the components have to be assembled.

Having a closer look at these modular furniture systems, one may note that

- a modular furniture system usually consists of a family of “related” pieces (they differ in size, colour, material, or shape),

- a component of one piece can be reused in a related piece,
- all pieces of the same family (or even of several independent families) only contain a small set of standard connecting elements,
- the components of a piece have to be assembled in a predefined way (there might be the possibility to have a restricted number of variations; e.g., the shelves of a book shelf can be placed in different heights), and
- it is usually possible to assemble all elements of a piece quite easily, without knowing i) what the components are built of and ii) how the components were built.

Another domain where the term architecture can be applied to are stereo (Hi-Fi) systems. A stereo system, for example, consists of an amplifier, CD-player, tuner, tape deck, and other components. Each of these stereo components is built up from smaller components (e.g. circuits), which again use even smaller components (e.g. transistors). All stereo components have a well-defined basic behaviour and support standard interfaces. Before they can be used, they have to be connected to a power supply. Although in principle each of these components would function by itself, their real value lies in the way they are designed to be plugged together.

A customer composing a stereo system is usually not interested in how the components are built, but is interested in the services they deliver (a tape deck should support a specific noise reduction system) and their composability (it should be possible to connect components from different vendors). The producers, on the other hand, have a different view of their hardware components: they know the internal architecture (design and implementation) of their components, and often reuse existing layouts and pieces of hardware to develop new components.

In fact, vendors have been building with standardized components for years, as they generally do not design a single stereo component, but rather design a *family* of them. Variations are made by combining a basic set of components into different configurations, and only a few components are made specifically for a single product.

As a last introductory example, consider the following Bourne Shell script [Bou78]:

```
grep -h '^#Keys' * | tr -c '[A-Z][a-z]' '[\012*]' | \
grep -v 'Keys' | sort -u | comm -13 keywords -
```

The purpose of this script is to extract keywords of a set of files and check whether they are contained in a predefined set of keywords (stored in a special file of keywords `keywords`).[1] In order to simplify the following discussion, we restrict the Bourne Shell and only allow the operators '|', '>', and '>&'.

Analyzing this Shell script, it is possible to find several interesting properties:

- the script consists of 5 components (`grep`, `tr`, `grep`, `sort`, and `comm`), each fulfilling a well-defined task,
- the components have to be instantiated,

[1] The usage and arguments of the UNIX programs used can be found in an appropriate UNIX manual.

- the functionality of the components can be specialized at instantiation by passing different (command line) arguments (e.g. `grep` is used with the arguments `-h '^#Keys' *` and `-v 'Keys'`, respectively),
- successive components are connected by a pipe (indicated by the '|' symbol) and interact using text streams,
- none of the components knows it predecessor and successor,
- the components and the pipes of the script form a pipeline, where each component depends on the output of its predecessor and the parameters used to instantiate it.

Analyzing the three examples mentioned previously in this section, we can deduce the following properties:[2]

- the architecture of a system or subsystem can be described in terms of *components* and *connecting elements* (connectors),
- a component fulfills a well-defined task and has a compositional interface,
- a connector connects a set of components by using their compositional interface,
- components and connectors cannot be connected in an arbitrary way; a set of rules restricts their compositions,
- a component can itself be a composition of (smaller) components and connectors,
- a component can be reused in a different context than the one it has been designed for,
- the type of components and connectors and the set of rules define the set of possible systems which can be built, thus defining an *architectural style*.

All the properties listed above hold for any kind of architecture, software architectures in particular. However, there are other properties and aspects which need to be addressed while discussing software architectures. A detailed discussion of these aspects is the topic of section 3.3. As a first summary, we consider an architecture as a description of a system in terms of components and connectors.

Although we have only informally introduced the term architecture, we already focussed on a specific view what we consider an architecture. The reader should note that the term is also used in a much broader sense in different areas. Examples of other senses are module architecture, configuration architecture, file architecture, process architecture etc.; refer to [SG96, chapter 1] or [DW99, chapter 12] for details.

We mentioned above that an architectural style restricts the set of possible systems. In order to illustrate this fact, reconsider our Bourne Shell script mentioned earlier in this

[2]The list only contains those properties which are relevant for architectures in general; other properties are omitted.

section, where we compared the keywords found in a set of files with a special file of keywords. The comparison only works since the file of keywords is already in a format suitable for the filter `comm`[3] (i.e. a text file with each keyword on a separate line). If the list of predefined keywords is not available in this format (e.g. keywords separated by semicolumns), it is necessary to create the appropriate format (by using a suitable helper application), which is usually not a major problem on its own. The problem lies in the nature of the filter `comm`: it is possible to use it with either two input files (`comm file1 file2`) or one file and one input stream (`comm file -`),[4] but not with two input streams. Therefore, it is necessary to store the list of predefined keywords in a temporary file, before we can use the script above. In order to avoid this temporary file, we would need i) a comm-like filter which is able read from two input streams and ii) a mechanism to connect the output streams of two filters to a single component.

The two requirements mentioned in the last paragraph cannot be fulfilled by our restricted Bourne Shell. This shell can be seen as a kind of compositional environment supporting a restricted set of components and connectors only. The main restrictions are that

- components (filters) have one input port (the standard input stream) and two output ports (the standard output and standard error streams),
- the behaviour of filters can be specialized at instantiation by passing different (command line) arguments, but no run-time reconfiguration is possible,
- connectors connect either i) the standard output or the standard error stream of a filter to a file, or ii) the standard output stream of a filter to the standard input stream of another filter, and
- it is not possible to introduce feedback loops between filters.

Since all shell scripts have to fulfill the restrictions mentioned above, they all share a similar overall structure. Or in other words: they conform to a *pipe and filter* architectural style [AAG93].

3.2 Roots of software architecture

The study of software architecture is in large part a study of software structure that began in the late sixties when Dijkstra pointed out that it pays to be concerned with how software is partitioned and structured, as opposed to simply programming so as to produce a correct result [Dij68]. He was the first author to describe a structure in which programs were grouped into layers, and programs in one layer could only communicate with programs in adjoining layers, resulting in major gains for development and maintenance.

[3] In UNIX terminology, an command like `comm` is usually called a *filter*.

[4] – indicates to use the standard input stream for comparison.

In programming, the term architecture was first used to mean a description of a computer system that equally applied to more than one actual implementation. Brooks and Iverson called architecture the "conceptual structure of a computer...as seen by the programmer" [BI69], making the distinction between architecture and implementation: "Where architecture tells what happens, implementation tells how it is made to happen." This distinction has survived until today, particularly in the area of object-oriented programming.

Parnas continued these investigations, resulting in the definition of so-called *program families* [Par76]: a program family is a set of programs for which it is profitable or useful to consider them as a group, and can be described by specifying a decision tree that was or would be traversed in order to arrive at each member of the family. The significance of the program family concept to software architecture is that a software architecture embodies those decisions at or near the top of Parnas' program family decision tree.

Alexander was the first author who tried to record design knowledge in a canonical form [AIS77]. He applied so-called *patterns*[5] in the context of the architecture of buildings and showed how patterns can be applied to house construction and the planning of neighbourhoods and whole cities. In the late eighties, software engineers recognized the importance of patterns in the area of software engineering and tried to adapt Alexanders approach, resulting in several publications [Gam91, Joh92, GHJV95, Pre95, BMR$^+$96] and a booming field of software engineering generally known as *design patterns*.

Although there are many ways of describing design patterns, the essential point is that they describe the solution of a recurring design problem in relationship to its context. The solution usually is a kind of generic *micro-architecture*, which can be applied in a context where the corresponding problem occurs. However, Buschmann et al. noted that patterns can also be applied to larger-scale problems and introduced the term *architectural pattern*:

> An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them [BMR$^+$96].

Patterns have several important contributions for the area of software architecture:

- patterns provide a common vocabulary and understanding for design principles, facilitating the discussion of design problems and their solutions [GHJV95],
- patterns identify and specify abstractions that are above the level of classes, instances, and components and, therefore, are a kind of template for concrete systems,
- patterns are a means of documenting software architectures [Joh92], and
- patterns help to build complex and heterogeneous software architectures.

[5] For Alexander's definition of pattern, refer to [Ale79].

Today, the study software architecture has emerged into an important field for both software engineers and researches. There is considerable activity in this area, which can be roughly placed into four categories, each addressing different aspects and problems [SG96]:

- codification of architectural expertise: cataloguing and rationalizing of architectural principles and patterns (refer to section 3.4),
- architectural characterization by (new) architectural description languages (refer to section 3.5),
- development of architectural frameworks for specific application domains, and
- development of formalisms for reasoning about architectural designs.

3.3 Definitions of software architecture

In the previous sections, we have used the term software architecture without giving a precise definition. Unfortunately, there is no generally-accepted definition of the term. In the following, we will summarize definitions and comments of other researchers, and will elaborate our own definition of software architecture, which will be used throughout the rest of this work.

One of the earliest definitions of the term software architecture was elaborated by Perry and Wolf [PW92]. Their goal was to build a foundation for studying software architectures. By analogy to building architecture, they propose a model of software architecture as a combination of *elements*, *form*, and *rationale*:

> A *software architecture* is a set of architectural (design) elements that have a particular form. Properties constrain the choice of architectural elements whereas rationale captures the motivation for the choice of elements and form [PW92].

Architectural elements are divided into *processing elements*, *data elements*, and *connecting elements*. The processing elements are those components that supply the transformation on the data elements. The data elements are the elements that contain the information that is used and transformed whereas the connecting elements are the glue that holds the different pieces of the architecture together. The architectural form consists of properties and relationships: properties are used to constrain the choice of architectural elements, and relationships constrain how the different elements may interact and how they are organized with respect to each other. Finally, the rationale explicates the satisfaction of the system constraints, which are determined by considerations ranging from functional aspects to various non-functional aspects such as performance and reliability.

A similar approach is used by Lea for the definition of flow patterns [Lea96], where he distinguishes between *representational components*, *transformational components* (stages), and *coordination components*.

Another definition of the term software architecture is given by Luckham and Vera in the context of RAPIDE, a concurrent event-based simulation language for defining and simulating the behaviour of systems at an architectural level:

> A *software architecture* is an executable specification of a class of systems, consisting of *interfaces*, *connections*, and *constraints*. The interfaces specify the behaviour of components of the system, the connections define the communication between components, and the constraints restrict the behaviour of the interfaces and connections [LV95].

Since the interaction between system components is defined by connections between their interfaces, Luckham and Vera call such an architecture an *interface connection architecture*.

A step further than Luckham and Vera goes the definition of Bass, Clements, and Kazman, introducing the concept of *visible properties*:

> The *software architecture* of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them [BCK98].

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. The intent of this definition is that a software architecture must abstract away some information from the system, but still providing enough information in order to be a basis for analysis, reasoning, and implementation.

In their definition, Shaw and Garlan introduce the notion of *recursive composition*:

> *Software architecture* involves the description of elements from which software systems are built, interactions among these elements, patterns that guide their composition, and constraints on these patterns. In general, a particular software system is defined in terms of a collection of components and interactions among these components. Such a system may be used as an element in larger systems [SG96].

Similar to the definition of Perry and Wolf, an architecture describes a software system in terms of computational elements and interactions among these elements, defining a kind of architectural model. Individual elements of an architectural model are defined independently, so that they can eventually be reused in different contexts. This view of an architecture is also applied by Magee et al. in the context of Darwin, a programming language for structuring parallel and distributed programs [MDK92].

Gacek et al. claim that most definitions of software architecture focus on what can be seen "from the street", but do not completely address the full range of evaluation issues associated with a software architecture:

> A *software system architecture* comprises i) a collection of software and system components, connections, and constraints, ii) a collection of system stakeholders' need statements, and iii) a rationale which demonstrates that the components, connections, and constraints satisfy the collection of system stakeholders' need statements [GAACB95].

Their main concern is the role of software architectures in the software life-cycle. The fact that different stakeholders[6] (customers, users, system engineers, developers, maintainers etc.) take different viewpoints when expressing their concerns about a software system implies that software architectures are involved in all phases of the life-cycle and require to consider more than just one view of a system. Each view (e.g. conceptual view, module view, process view, physical view) reflects a specific set of concerns that are of interest to a group of stakeholders.

The definition of Buschmann et al. considers the impact of functional and non-functional properties on the development of a system, and the importance of *views* in order to show the properties of a system:

> A *software architecture* is a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and non-functional properties of a software system [BMR+96].

According to their definition, a component is an encapsulated part of a software system which has an interface and serves as a building block for the structure of a system. A relationship denotes a (static or dynamic) connection between components whereas a view represents a partial aspect of an architecture that shows specific properties of a software system.

Finally, based on structures found to be prevalent and influential in the development environment of industrial projects, Soni et al. identify four distinct categories of architectures, each describing a system from a different perspective [SNH95]:

- the *conceptual architecture* describes the system in terms of its major design elements and the relationships among them,
- the *module interconnection architecture* encompasses two orthogonal structures: functional decomposition and layers,
- the *execution architecture* describes the dynamic structure of a system,
- the *code architecture* describes how the source code, binaries, and libraries are organized in the development environment.

Although the definitions mentioned in this section focus on different aspects, they all have one important aspect in common: making a clear separation between computational

[6] A stakeholder is a person who has a financial interest in the system.

elements and their relationships. Summarizing the definitions, we get a similar set of properties as the ones described in section 3.1. However, the following properties have to be added to the list:

- a software architecture describes a system at a level beyond simple algorithms and data structures, including global organization and control structure,
- the architecture of a system consists of computational elements (components), data elements, and connecting elements (connectors). The data elements are often implicitly described in the connecting elements,
- components can be described at different levels of abstraction (e.g. a component may be as small as a single procedure or as large as an entire application), and
- different views and levels of architectural abstraction are needed in order to describe specific aspects and properties of a system.

Some of these properties are essential for the description of a software system (e.g. components and connectors), other properties give additional information about the context and rationale of an architecture. It is obvious that the latter properties are important for documentation and define guidelines how a system can evolve and be modified, but only the former properties are essential when a system has to be implemented the way it is described.

In order to define the term software architecture for our purpose, reconsider the definition of a component given in section 2.2: a component is a black box entity with a well-defined set of required and provided services. It is obvious that connections have to be established between required and provided services of the components of a system. Otherwise, the components would not be able to cooperate with each other. This observation leads to the following definition of the term software architecture:

> A *software architecture* describes a software (sub-)system as a configuration of components and connectors. A connector connects required ports of a set of components to provided ports of other components. A configuration of components and connectors can be used as a component of another (sub-)system.

This definition has several consequences. First of all, it does not restrict the level of abstraction of components and connectors: they can be described at different levels of abstraction, where each level shows different aspects of the architecture of a system. Second, a component may itself be a composition of (smaller) components and connectors. Third, the behaviour of a (sub-)system is specified by the properties of its components and the corresponding connections; the notion of the architecture itself does not add any new behaviour. Fourth, if the properties of all components and connectors of a configuration are known, it is possible to analyze an architecture at an appropriate level of abstraction. This is important if it is necessary to check whether the connections between component ports are well-formed, which might lead to the development of a kind of architectural type system. Finally, a software architecture can be seen as a template for implementation: every implementation which fulfills all the properties of the architecture is an instance of the corresponding architecture. However, although the architecture does not make any explicit assumptions about the programming language to be used for implementation, it restricts the set of possible programming languages due to the fact that certain component types used in the architecture (e.g. classes) may not be present in any language.

Note that the term architecture is still used today in some communities to refer to the user view of a system [Kru95], but this is not what is meant by our definition of the term.

3.4 Architectural styles and patterns

Often a software architect is not concerned with a single system in isolation, but with a system in the context of an entire family of systems. In such a situation, a software architect usually does not design a system from scratch, but searches for an already existing software system that has solved a similar problem or a problem in the same application domain. Thus, the architect tries to reuse and adapt an architecture from a particular *architectural style*. By doing so, it is guaranteed that the resulting architecture has some desirable properties, and allows him to use a vocabulary that is natural to the problem domain and supports reasoning and analysis about specific aspects of the system.

But what precisely is an architectural style? Again, no generally-accepted definition of the term exists. Therefore, we will summarize definitions of other researchers and elaborate a definition for our own purpose.

3.4.1 What are architectural styles?

Perry and Wolf consider an architectural style as an arrangement which abstracts elements and formal aspects from various specific architectures:

> An *architectural style* defines a family of software systems in terms of their structural organization. An architectural style expresses components and the relationships between them, with the constraints of their application, and the associated composition and design rules for their construction [PW92].

Given this definition, they claim that there is no hard dividing line between an architecture and an architectural style. The important thing about an architectural style is that it encapsulates important decisions about architectural elements and emphasizes constraints on elements and their relationships.

Shaw and Garlan define an architectural style as a vocabulary of components, connectors, and constraints:

> An *architectural style* defines a family of systems in terms of a pattern of structural organization. More specifically, an architectural style defines a *vocabulary* of components and connector types, and a set of *constraints* on how they can be combined [SG96].

According to Bass, Clements, and Kazman, an architectural style consists of a few key features and rules for combining those features so that *architectural integrity* is preserved:

> An *architectural style* is a description of component types and a pattern of their run-time control and/or data transfer. It is determined by i) a set of component types, ii) a topological layout of these components (indicating their run-time interrelationships), iii) a set of semantic constraints, and iv) a set of connectors that mediate communication, coordination, or cooperation among components [BCK98].

Depending from the point of view, an architectural style either defines a class of architectures or is an abstraction for a set of architectures that meet it.

Summarizing and combining the principle ideas of the definitions above, we define the term architectural style as follows:

> An *architectural style* is an abstraction over a set of related software architectures. It defines a vocabulary of component and connector types and a set of rules how components and connectors can be combined.

In contrast to the fact that each software system has an architecture, it is often not possible to assign a single architectural style to a system: a system may be a combination of several architectural styles. Such systems are called *heterogeneous* [SG96, BCK98]. Bass, Clements, and Kazman distinguish between three kinds of heterogeneity:

- *locally heterogeneous*: the architectural structure of a system reveals different styles in different areas (e.g. branches of a main-program-and-subroutines system[7] have a shared data repository),
- *hierarchically heterogeneous*: a component of one style, when decomposed, is structured according to the rules of a different style (e.g. an element of a pipe and filter pipeline is structured in a layered style), and
- *simultaneously heterogeneous*: a system may be described using several architectural styles.

The last form of heterogeneity implies that it is not possible to partition architectural styles into nonoverlapping categories.

3.4.2 Classifying architectural styles

The use of architectural styles has several advantages as they incorporate the knowledge of experienced software architects. However, before we can benefit from this knowledge, it is necessary to establish a vocabulary of architectural styles and define a corresponding

[7] For a description of the *main-program-and-subroutines* style and other styles mentioned in this section, refer to [BCK98, chapter 5].

classification, similar to the design pattern catalogue by Gamma et al. for the solution of finer-grained design problems [GHJV95]. Such a classification would facilitate the communication between different people involved in the design of a system, help a software architect to find the appropriate style for a given problem, and promote design reuse since solutions with well-understood properties can be reapplied to new problems (refer to section 3.6 for a discussion about the influence of software architectures on reuse).

A first attempt to identify and define architectural styles has been proposed by Abowd et al. [AAG95] and Shaw and Garlan [SG96]. However, both approaches were restricted to only a small set of architectural styles and did not cover a wide range of styles from different problem domains. An attempt to classify a broader range of architectural styles has been elaborated by Shaw and Clements [SC97] and Bass, Clements, and Kazman [BCK98]. Common to all approaches is that they do not capture all specializations and modifications of given styles found in the software world. The styles are described in a very general way and leave many design decisions open. Refinements to those styles are necessary before they can be effectively used to build real software architectures.

A detailed discussion about existing approaches to classify architectural styles is beyond the scope of this work; please refer to the corresponding references. For an overview of the catalogue proposed by Bass, Clements, and Kazman, particularly the refinement of the *data flow* style, refer to [BCK98, chapter 5].

3.4.3 Architectural patterns

In section 3.2, we have discussed architectural patterns and their contributions to the area of software architectures. If we compare this discussion with the definitions of architectural style, we see that there are many aspects in common. Often both terms are used simultaneously, although we think that there is a distinction between an architectural pattern and an architectural style: *an architectural pattern is a way to describe an architectural style*, including more information than just a vocabulary of component and connector types and a set of rules. Generally speaking, an architectural pattern expresses a fundamental structure for a software system with an associated method that specifies how to construct it. An architectural pattern comprises information about when to use it, its invariants and specializations, as well as the consequences of its application.

> An *architectural pattern* describes the solution of a particular recurring design problem that arises in specific design contexts. The solution scheme describes the overall structural organization (components, their responsibilities, and the relationships between them), the constraints of its application, and the associated composition and design rules.

This definition of the term *architectural pattern* is very similar to the definition of an architectural style by Perry and Wolf [PW92]. We believe, however, that it is important to make a clear distinction between separate concepts and introduce well-defined terms for each concept. Therefore, we make the distinction between an architectural style and an architectural pattern: the former focuses on the technical aspects (vocabulary of components and connectors, composition rules) of the solution scheme whereas the latter describes the associated design rules, the consequence of its usage, and related solutions.

3.5 Architectural description languages

Often, software architectures are represented by box-and-line diagrams in which the properties of components, the semantics of connections, and the behaviour of the system as a whole are poorly defined. Even though such diagrams give a first, intuitive picture of a system, they are highly ambiguous and are a source of misinterpretation: it is virtually impossible to answer many questions that are important for the clear understanding of a system. What we need is a notation to precisely describe software architectures and architectural styles. In the following, we call such an notation an *architectural description language* (ADL):

> An *architectural description language* is a notation that allows for a precise description and analysis of the externally visible properties of a software architecture, supporting different architectural styles at different levels of abstraction.

The questions arises what kind of requirements a notation needs to fulfill in order to be useful as an architectural description language. According to Allen [All97], an ADL must offer a vocabulary that can be easily understood by architects and is suitable for communicating architectures to all interested parties of a system. Second, it must directly provide the abstractions of components and connectors. Third, an ADL must provide a precise semantics for components and connectors that resolves ambiguity and aids in the detection of inconsistencies. Finally, it must offer a set of techniques that facilitate analysis of architectural styles and support reasoning about specific system properties.

Focusing primarily on linguistic issues, Shaw and Garlan have elaborated the following concepts that an architectural description language must provide [SG96]:

- *abstraction*: describing components and their interactions of a software system with clear and explicit abstract roles,
- *composition*: describing a system as a composition of independent components and connectors,
- *reusability*: reuse components, connectors, and architectural patterns in different architectural descriptions,
- *configuration*: localization of system descriptions (independent of elements being structured); dynamic reconfiguration of systems,
- *heterogeneity*: combining multiple, heterogeneous architectural descriptions,
- *analysis*: performing analysis of and reasoning about architectural descriptions.

Bass, Clements, and Kazman add additional requirements for architectural description languages [BCK98]. For our purpose, an ADL must

- support the common architectural styles,
- enable the creation, refinement, and validation of architectures,
- provide a basis for further implementation.

This covers most of the requirements for architectural description languages, and more or less conforms to the requirements described by Luckham and Vera for the design of RAPIDE [LV95]. However, some of the requirements mentioned above need to be refined. For example, an ADL must be open enough to support new architectural styles (additional component and connector types, new composition rules etc.).

There is a large variety of architectural description languages emerging from either industrial or academic research groups. A detailed comparison of these languages is beyond the scope of this work. For further discussion of this topic, refer to [SG96] or [MT97].

3.6 Influence of software architectures on software engineering

The use of software architectures has an influence on all the phases of the software life-cycle and, therefore, is an important issue in the entire software life-cycle process [GAACB95]. In this section, however, we will not discuss the influence of software architectures on the entire software life-cycle, but focus on those aspects which are important for reuse. For a discussion of all the other aspects, refer to the corresponding references (e.g. [GAACB95, SG96, BCK98]).

The use of software architectures and architectural styles has several benefits in the area of reuse. Before we discuss selected aspects in further detail, consider three statements which directly address the problems of reuse related to architectures and components.

> The important lesson in reusing components is that the possibilities for reuse are the greatest where specifications for the components are constrained the least – at the architectural level. Component reuse at the implementation level is usually too late [PW92].

According to Perry and Wolf, the primary focus of software architectures is the identification of important properties and relationships – constraints on the kinds of components that are necessary for the architecture, design, and implementation of a system. Component reuse at the implementation level is often too late because the implementation elements embody too many constraints.

> Having a reusable architecture is a precondition to successfully developing reusable components [BCK98].

Reusable architectures provide the structure and coordination model for a family of systems whereas reusable components implement basic concepts used in these systems.

> A *component framework* is a dedicated and focused architecture, usually around a few key mechanisms, and a fixed set of policies for mechanisms at the component level [Szy98].

If we rephrase this statement in the terms we have introduced previously in this chapter, we see that a component framework implies a specific (often restricted) set of architectural styles, offering generic implementations for basic components and connectors.

We can deduce from the statements above that the use of software architectures is one of the key concepts for developing reusable components. Some of the main implications of software architectures for reuse are:

- components can be reused easier at a level of abstraction where they are less constraint than at a level with more constraints,
- before a component framework can be developed, its is necessary to have a clear understanding of the architectural styles it supports and the basic components and connectors it implements,
- the use of a component framework restricts the architecture of a system to be implemented (i.e. a framework cannot be used if a system architecture violates these restrictions), and
- a new component can only be integrated into a framework when the framework supports architectural styles with the corresponding component types.

After investigating the influence of software architectures on reuse, we argue that software architectures and architectural styles not only have an influence on reuse at *design level* (design reuse), but also at *implementation level* (code reuse).

Design reuse. The use of architectures and architectural styles is a different approach to develop software. As mentioned in section 3.4, it is often not desirable to design a system from scratch, but to look for already existing software systems that have solved a similar problem or a problem in the same application domain. Reusing the architecture of such systems has the advantage that the new system benefits from well-understood properties and important design decisions of existing systems.

If the corresponding problem domain matures, people tend to develop reusable reference models and architectures: reference models describe specific configurations of components and interactions for specific application areas [SG96]. As an example, a compiler is usually divided into a lexer, a parser, a semantic analyzer, an optimizer, and a code generator. A consequence of reference models is the development of so-called *architectural frameworks*. An architectural framework determines the structure of a family of applications, providing shared infrastructure and prescribing requirements for instantiating the framework to produce a particular application [AGI97]. In contrast to a component framework, an architectural framework is specified at a higher level of abstraction, using an architectural description language. An architectural framework precisely describes common reference architectures for a family of systems, and helps in the development of architectural standards.

Reusing the architecture of existing systems often allows a developer to use a vocabulary that is natural to the corresponding problem domain, and may lead to the identification of reusable named architectural abstractions (e.g. architectural patterns). As already mentioned in section 3.2, patterns identify and specify abstractions that are above the level of classes, instances, and components, defining reusable micro-architectures (with desirable properties) which can be used for solving specific design problems. In contrast to reference models, patterns are usually not restricted to specific application domains and, therefore, it is possible to use them for a broader range of design problems.

Code reuse. An often underrated aspect is the influence of architectures on code reuse. It is often easier to implement a particular system using a component framework which already supports the architectural styles the system is designed in than implementing the system from scratch, even when not all the required components are part of the framework. As an example, when a system is designed using a pipe and filter architectural style, the corresponding implementation can certainly profit from components and connectors offered by the Bourne Shell. Such an approach also enables a software engineer to reuse existing components offered by the framework, maybe with some additional adaptation to precisely fit the requirements of the system.

If a software engineer is concerned with the design and implementation of specific subsystems for a family of related systems, he is encouraged to develop and implement components in a way that they can be reused for all systems of the family. Such a reuse-oriented development can lead to the development of a small component framework, implementing standard components and basic connectors, tailored to fulfill the requirements of the corresponding subsystems. Such small component frameworks are sometimes referred to as *framelets* [PS99].

There are aspects which cannot be assigned to one of the two categories discussed above, but have an influence on both design and code reuse. As an example, consider the area of reengineering and reverse engineering of legacy systems. Such systems are often badly documented (if at all) and implemented in out-of-date technologies. In order to extend and adapt these systems, it is essential that the underlying architecture is known or can be extracted using specialized analysis techniques. Once the architecture is known, it can be adapted to new requirements. Depending on the implementation technology of the legacy system, the new system can reuse parts of the legacy implementation. Therefore, legacy system have an influence on design and code reuse. However, efficient reengineering of legacy systems is still an art, especially in the area of object-oriented legacy systems and, therefore, a field of active research [DDN$^+$97].

Finally, we would like to point out to a particular reuse problem we will discuss in chapter 5: *architectural mismatch*. Architectural mismatch stems from mismatched assumptions a component makes about the structure of the system it is to be part of [GAO95]. In other words: it is not possible to reuse a component in a software system which does not conform to the architectural styles supported by the underlying component framework or composition system.

3.7 Related aspects

When designing a large, complex software system, it is often necessary to consider more than just one structural perspective of the system. If system properties such as physical distribution or synchronization must be characterized at an architectural level, then these properties are best addressed by using different *architectural views*. A view represents a partial aspect of a software architecture that shows specific properties of a system [BMR$^+$96]. Therefore, a view can be seen as an abstraction over a software architecture which focuses on specific aspect of interest. Views are not fully independent of each other, as elements in one view can relate to elements in another view and, therefore, it is necessary to reason about the interrelations of these views.

There is not yet an agreement on a standard set of views or terms to refer to views. Soni et al. [SNH95] proposed four distinct categories of views to describe a software system (refer to section 3.3). A similar approach is taken by Clements and Northrup, where the following views are distinguished [CN96]:

- *conceptual (logical) view:* this view depicts functional requirements of a system at an abstract level,
- *module (development) view:* this view focuses on the organization of actual software modules,
- *process (coordination) view:* this view focuses on the run-time behaviour of a system, and
- *physical view:* this view considers mappings of the software onto hardware.

An important step towards a more scientific approach is the introduction of appropriate *formal foundations for software architecture*, and it is generally agreed that formal models and techniques are important for a mature engineering discipline. In the field of software architectures, formalisms can be useful in order to provide precise, abstract models, for describing specific designs, and for simulating behaviour. Shaw and Garlan identify four different uses of formalisms [SG96]:

- *the architecture of a specific system:* formalisms of this kind precisely specify all the properties of a specific system.

- *an architectural style:* formalisms of this kind can be used to describe architectural abstractions for families of systems.

- *a theory of software architecture:* formalisms of this kind clarify the meaning of generic architectural concepts, such as architectural connection, hierarchical architectural representation, and architectural styles.

- *formal semantics for ADLs:* this kind of formalisms apply techniques to formally specify the semantics of architectural description languages.

For a detailed discussion of existing formalisms for software architectures, refer to the Ph.D. thesis of Allen [All97].

Besides software architectures, there exist other approaches to describe the structure of software systems. A number of so-called *Module Interconnection Languages* (MILs) have been developed to support the description of large-scale program structure independent of programming languages. MILs define program structure through definition/use bindings that associate definitions of program constructs (e.g. data structures, functions) with uses of those constructs. In addition, they also provide an explicit and separable description of a system's structure by explicitly binding definitions to uses. However, MILs do not provide a means of specifying patterns of interaction or the development of families of systems.

To overcome some of the restrictions of MILs, *Module Interconnection Formalisms* (MIFs) have been developed. They support the composition of software modules based on mechanisms other than definition/use bindings, and define interconnection semantics. MIFs provide a set of interaction abstractions as the basis for system composition. As an example, POLYLITH provides run-time support for message passing between components based on a *software bus* [Pur94]. All communication is mediated by this bus, isolating data transformations and physical distribution from component implementations. MIFs support flexible implementation of systems that explicitly use the interaction mechanisms provided by a particular implementation. However, MIFs cannot be used to build systems that use other interaction mechanisms or communication patterns.

A common problem for the integration of systems is the potential for inconsistency in data representations. In order to address this issue, *Interface Definition Languages* (IDLs) have been developed. IDLs define abstract data interfaces and mappings from these data interfaces to programming language constructs for languages with a corresponding IDL binding. Examples of interface definition languages are the IDLs of CORBA [OMG96] and COM [Rog97]. Using IDLs, incompatibilities between modules arising from data structure representations and differences in implementation platform can be resolved. Like MIFs, IDLs support a single, restricted form of interaction. They define the data that is passed between components, but the dynamic constraints are either implicit in the language mechanisms underlying the IDL or remain unspecified.

3.8 Summary

In this chapter, we have given a general introduction to software architectures, analyzed definitions of other researchers, clarified the corresponding terms, and discussed selected aspects related to software architectures. In this section, we will summarize the main observations of this chapter. It is obvious, however, that there are other aspects of software architectures which are also relevant (e.g. quality attributes, architectural erosion), but are beyond the scope of this work.

A *software architecture* describes a software (sub-)system as a configuration of components and connectors. A connector connects required ports of a set of components to provided ports of other components. A configuration of components and connectors can be used as a component of another (sub-)system. The main purpose of software architectures is to make a clear separation between computational elements (components) and their relationships (connectors). An *architectural style* is an abstraction over a family of software architectures. It defines a vocabulary of component and connector types and a set of rules defining how components and connectors can be combined.

An *architectural description language* (ADL) is a notation that allows for a precise description and analysis 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.

The use of software architectures and architectural styles has an influence on all phases of the software life-cycle, but most importantly they are one of the key concepts for developing reusable components and frameworks.

Chapter 4

Scripting

In the previous chapter, we have discussed the influence of software architectures on component frameworks. Naturally, it is not enough to have components and frameworks, but for building real applications one needs a way to wire components together (i.e. to express *compositions*).

In recent years, so-called *scripting languages* have become increasingly popular as they make it very easy to quickly build small, flexible applications from a set of existing components. These languages typically support a single, specific architectural style of composing components (e.g. the pipe and filter architectural style supported by the Bourne Shell), and they are designed with a specific application domain in mind (system administration, graphical user interfaces etc.). It turns out that scripting languages do not only allow for flexible composition, adaptation, and configuration of existing components, but also to represent higher-level design elements in applications, which makes them ideal tools for expressing applications as compositions of software components.

In this chapter, we give a general introduction into scripting and analyze the concept of scripting as the key mechanism for composing components. We discuss the main properties and abstractions of scripting languages, and identify a list of essential and characterizing features. We compare selected scripting languages, illustrate some important concepts of each of these languages, and analyze abstractions for executing dynamically created code (also known as "eval" feature). We conclude this chapter with a brief summary of the main observations.

4.1 What is scripting?

Similar to software architectures, there are no generally-accepted terms for scripting and scripting language. Existing definitions are rather vague at the best and characterize scripting languages somewhere between

> A scripting language is a language that is primarily interpreted, and on a UNIX system, it can be invoked directly from a text file using `#!`.
>
> *Anonymous Usenet User*

and

> A scripting language introduces and binds a set of software components that collaborate to solve a particular problem [NTMS91].

Therefore, we will again summarize definitions and comments of other researchers and elaborate our own definitions, which will be used throughout the rest of this work. Many of the cited definitions are the result of personal communications with the corresponding researchers and, therefore, cannot be assigned to specific references.

Scripting. Scripting is often associated with CGI-programming (e.g. Perl [WCS96]) or animating web pages (e.g. JavaScript [Fla97]). Both application domains are typical for scripting languages, and they reflect the main purpose of scripting technology:

> The essence of a "script" is a (usually) short program that "drives" some other programmable system (e.g. the UNIX operating system). *Guido van Rossum*

Although the main purpose of CGI-scripts is to dynamically generate web pages, they generally do not perform all the necessary computations themselves: they use other components on the server system to do the job for them. Most of the code of CGI-scripts initiates and coordinates (i.e. "drives") computations of these components. The same applies to animated web pages using JavaScript: the main purpose of scripts is to control and extend the behaviour of a web browser (control document appearance and content, interact with applets etc.), and not to fulfill major computations.

In the same direction, but a step further, goes the definition of Larid: a script acts as *glue* between components:

> *Scripting* labels a "high-level" language that gets something outside itself (a browser, system facilities etc.) to do the work of an application. Other metaphors that emphasize this role are "glue" and "bricks and mortar". Scripting typically involves "rapid development" with a notion of interpreted source execution, weak typing, and introspective facilities. *Cameron Larid*

Scripts can be seen as a kind of mortar "gluing" bricks (i.e. components) together. The essence is that the bricks are *outside* the scripting language and that the glue is at a *high level* of abstraction. This definition points out to further dimensions of scripting languages (interpretation, typing, introspection) which will be further discussed in section 4.2. The reader should note that Larid uses the term *glue* in a much broader sense than other references (e.g. [DW99]), and the definition we will give in section 5.1 is much more restrictive.

According to [Szy98], scripting is quite similar to application building. Approaches based on scripting admit that the actual *wiring* may need more than just connections: scripting allows small programs (i.e. *scripts*) to be attached to connections (i.e. connectors). This can either be a the source end (e.g. for events) or at the target end (i.e. hooks) of connections. Unlike mainstream component programming, scripts usually do not introduce new components, but simple plug existing ones together: scripts can be seen as introducing behaviour but no new state. Or in other words:

> *Scripting* aims at late and high-level gluing [Szy98].

Summarizing the properties of scripting mentioned above, we see that the main purpose of scripting is to build applications by connecting a set of existing components. Generalizing this notion, we define the term scripting as follows:

> *Scripting* is a high-level binding technology for component-based systems.

This definition describes the overall purpose of scripting, but it also implies that there are other binding technologies for component-based systems and applications. Furthermore, the definition does not characterize scripting in further details. This is the purpose of the next paragraph where languages for scripting are discussed and analyzed.

Scripting language. There are two major directions researchers use to characterize scripting languages: by their *usage* and by their *features*.

As mentioned above, the purpose of a script is to drive a programmable system and to establish connections between components. Therefore, one may argue that any programming language which supports these activities can be seen as a scripting language:

> I suppose one might draw the line by saying that a *scripting language* is one where the main effect of a program is to drive another system, while in a programming language the program itself is the main action.
> *Anonymous Usenet User*

It is important to note that the author of this definition makes the difference between scripting languages and "other" programming languages. As we will see further below, the category of these other languages can be further subdivided.

Into the same direction goes the following comment:

> We recognize *scripting languages* by their uses, not by counting a checklist of features. *Cameron Larid*

Although it is certainly important to characterize a language by the way it is generally used, we believe that the kind of features a language supports has a major impact on its usage. It is, therefore, not enough to just observe the main usage of a language, but to describe its features and characteristics, and to analyze *why* it is used in a specific way.

A first attempt to define the term scripting language is the following:

> A *scripting language* relies on a class/interface/type-based underlying object-oriented/component-based language and run-time system. It allows programmatic control of only instantiation of existing component types and messages to those instances. *Doug Lea*

This definition describes a scripting language as a language which is only able to instantiate and connect components. In particular, it excludes the definition of new abstractions (e.g. new component types) and, therefore, a scripting language can be seen as an impoverished prototype style object-oriented language (such as SELF [US87]).

The definition given above is very restrictive and important aspects of scripting languages are not covered. The following definition elaborates some of these aspects:

> A *scripting* language should i) have text processing primitives, ii) offer some form of automatic memory management, iii) not require a mandatory separate compilation phase, iv) favour high-level expressiveness over execution speed or ability to manipulate data at the bit level, and v) interface well with the rest of the system. *Guido van Rossum*

Although not emphasized, item v) denotes one of the most important properties of a scripting language: interfaces to the "outside world" (i.e. the possibility to interoperate with components not written in the scripting language itself). Another important property van Rossum mentions is item iv): performance and resource consumption are expected to be dominated by the components (and not the scripts) and, therefore, the performance of the scripting engine usually is not a major issue.[1] However, it is important that a scripting language offers high-level abstractions for connecting components. The items ii) and iii) describe implementation specific aspects, and are not inherently due to scripting problems in general. The same also applies to item i), which is a feature common to several scripting languages.

Van Rossum also points out that in practice, most typical scripting languages are crippled for the development of large programs, because certain features that are useful for scripting or throw-away programming tend to be harmful in large programs (e.g. automatic data coercions).

Other important aspects of scripting languages are covered by the following definition:

> A *scripting language* should i) be interpreted, not compiled, ii) be dynamically typed (so that a variable can have different types during its lifetime), iii) offer abstractions for introspection, iv) be embeddable and extensible, and v) have a simple syntax. *Brent Welch*

Embeddability and extensibility are two important properties of scripting languages, which tells them apart from other programming languages. Extensibility is needed in order to incorporate new abstractions (components or connectors) into the language, encouraging the integration of legacy code. Embedding a script into an existing component offers a flexible way for adapting and extending this component, and has a major impact on reuse.

[1] One should note that the run-time systems of scripting languages are not necessarily slower than the one of so-called system programming languages. For example, Perl is highly optimized for text processing, and for some applications it requires a lot of skill to write a corresponding program in C or C++ which runs faster than a corresponding Perl application.

In the context of making a clear separation between computational elements and their relationships, Griffel defines the term scripting as *application level programming* and claims that the main purpose of a scripting language is the adaptation, customization, and configuration of components [Gri98]. From his point of view, a scripting language is interpreted, at most dynamically typed, and has a small language model. In particular, a scripting language should not have any procedural elements, but be a purely declarative, rule-based language for a specific application domain.

An important reference about scripting and scripting languages is an article by Ousterhout [Ous98]. He makes the distinction between *assembly languages*, *system programming languages*, and *scripting languages*.

In an assembly language, virtually every aspect of a processor is reflected in a program. Each statement represents a single machine instruction, and programmers must deal with low-level details such as register allocation and procedure calling sequences.

System programming languages differ from assembly languages in two ways: they are *higher-level* and they are *strongly typed*. The term higher-level means that many lower-level details (e.g. register allocation) are handled automatically by the programming environment and are not reflected in the program code. It usually takes several instructions of an assembly language in order to get the same functionality as a single instruction in a system programming language. In the context of a system programming language, typing refers to the degree to which the meaning of information is specified in advance of its use. In a strongly typed language, a programmer declares how each piece of information will be used, and the programming environment prevents the information from being used in a different way. To be more precise: a programming language is strongly typed if it is possible to ensure that every expression is type consistent based on the program text alone [CW85]. The main purpose of system programming languages is to create applications (and components) starting from scratch.

Ousterhout defines the term scripting language as follows:

> *Scripting languages* are designed for gluing applications. They provide a higher level of programming than assembly or system programming languages, much weaker typing than system programming languages, and an interpreted development environment. Scripting languages sacrifice execution efficiency to improve speed of development [Ous98].

Ousterhout claims that scripting languages represent a different style of programming than system programming languages. They are neither intended for writing applications from scratch nor for implementing complex algorithms or data structures: they assume a collection of existing components and are intended for plugging these components together. Therefore, they are sometimes referred to as glue languages or *system integration languages*.

Scripting languages are higher-level than system programming languages in the sense that a typical statement of a scripting language executes several hundred machine instructions, much more than a typical statement of a system programming language. Much of this difference is because the abstractions of scripting languages have a greater functionality than abstractions in system or assembly programming languages.

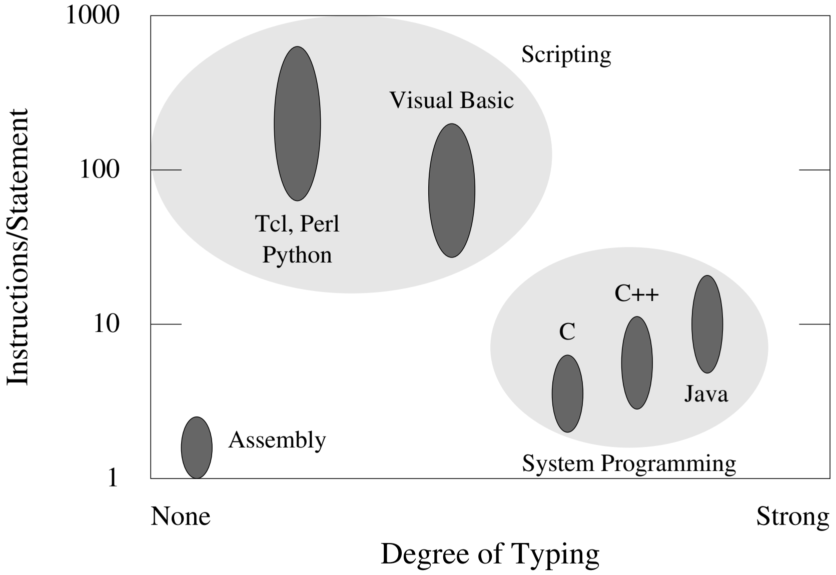


Figure 4.1: Comparison of programming languages based on their level of abstraction and their degree of typing (based on a similar comparison in [Ous98]).

In order to simplify the task of connecting components, scripting languages tend to be *weakly typed*.[2] A weakly typed language makes it easier to hook together components than a strongly typed language and allows us to reuse components in a way not foreseen by the designer.

For a graphical comparison of these three programming language categories (including sample languages), refer to Figure 4.1.

Analyzing the definitions given above, there remains one aspect which has not been explicitly pointed out to so far: scripting languages offer explicit support for architectural styles and representing high-level design elements in applications.

Due to the higher level of abstraction of a scripting language, it is possible to directly represent high-level design elements (i.e. high-level connectors) in applications, which implies an explicit support for software architectures. However, most existing scripting languages only offer a small set of such high-level abstractions and, therefore, support for a restricted set of architectural styles only. As an example, reconsider the discussion about Bourne Shell scripts in section 3.1. The Bourne Shell [Bou78] offers an operator '|' which is typically used to connect the standard output stream of one process to the standard input stream of another process, making the *pipe and filter* architectural style of a shell script explicit. One may note that in general it is not possible to use another architectural style in the Bourne Shell. It is obvious that we could also use a system programming language like C to instantiate and connect a set of UNIX processes. However, the C programming

[2]Some scripting languages or not typed at all, others are dynamically typed and/or offer run-time type conversions. For further discussion about typing issues, refer to section 4.3.

language does not offer high-level abstractions for connecting the standard output stream of a command to the standard input stream of another command and. Hence, it is not easily possible to make the architecture of the application explicit, especially as C does not offer such a convenient syntax for expressing connections of UNIX commands as the Bourne Shell.

Summarizing the properties discussed so far, a scripting language can preliminarily be characterized as follows:

- The purpose of a scripting language is the development of applications by plugging existing components together (i.e. the primary focus is on *composition*).
- Scripting languages are *extensible*: they are designed for extending the language model with new abstractions (e.g. new components and connectors) and for incorporating components written in other languages.
- Scripting languages are *embeddable*: it is possible to embed them into existing components, offering a flexible way for *adaptation* and extension.
- Scripting languages favour *high-level programming* over execution speed.
- Scripting languages are *interpreted* and offer *automatic memory management*.
- Scripting languages are *dynamically* and *weakly typed* and offer support for runtime *introspection*.
- Scripting languages offer *explicit support for architectural styles* (i.e. making the architecture of an application explicit), and can therefore be considered as executable architectural description languages (ADLs).

As we will discuss in the next section, not all of the features listed above are *essential* for scripting, but rather characterize a particular scripting language in the language space. Considering the essential properties, we define the term scripting language as follows:

> A *scripting language* is a high-level language used to create, customize, and assemble components into a predefined software architecture.

4.2 Dimensions of scripting languages

In the previous section, we pointed out that there does not exist a generally-accepted definition for scripting language and that existing definitions are rather vague at the best. Based on a summary of these definitions, we elaborated our own definition of scripting language, which defines the term at high level, but does not include any particular features a scripting language must support (i.e. any high-level language suitable for assembling components into a predefined architecture can be named a scripting language). In this section, we further discuss the summary of features elaborated in the previous section and divide the list of features into *essential* and *characterizing* features.

According to Ghezzi and Jazayeri, a programming language can be seen as a formal notation for describing algorithms for execution by a computer [GJ98]. In general, a programming language has two major components: *syntax* and *semantics*. The syntax is a set of rules that specify the composition of applications from letters, digits, and other characters. The semantic rules of a language specify the meaning of a syntactically correct program by mapping each language construct into a domain whose semantics is known (e.g. a mathematical domain like the λ-calculus). Programming languages are generally not categorized according to their syntax or their semantic domain, but according to the language constructs (or *features*) they offer (or do not offer). Depending on the provided features, a programming language is assigned to a programming *paradigm* such as imperative, functional, object-oriented etc. Moreover, a programming language can belong to multiple programming paradigms: C++, for example, supports both an imperative and object-oriented style of programming. Such programming languages are called *hybrid* whereas languages belonging to only one paradigm are called *pure*.

A programming paradigm defines a set of features which must be supported by any language belonging to the paradigm. For the rest of this work, we denote this set of features as *essential*. On the other hand, there are features which some (or many) languages of a particular paradigm support, but are not essential. As an example, many object-oriented programming languages have a root-class which is an (often implicit) ancestor to all other classes (e.g. `Any` in Eiffel, `Object` in Java). However, there are programming languages that do not have such a feature (e.g. C++), but are still considered to be object-oriented. We denote such features as *characterizing* (since they characterize a programming language in the language space). The list of characterizing features should be orthogonal in the sense that none of the features can be expressed as a combination of other features.

4.2.1 Essential features

After an analysis of the main features of scripting languages and the requirements for composing applications using components and scripts, we have identified two concepts which are essential for a scripting language: i) *encapsulation and wiring* and ii) *a foreign code concept*. These two concepts are the topic of the following section.

Encapsulation and wiring. In order to build an application as a composition of components, a scripting language must support some notion of components and connectors. More precisely, a language must offer mechanisms for encapsulation and wiring.

In section 2.2, we have defined the term component as a black-box entity with a well-defined set of required and provided services. Therefore, a language must define a set of primitives that correspond to such a definition.[3] This implies that it is possible to

[3] In the following, we will denote these primitives as built-in *component types*.

encapsulate all services of a component and define the interface in a way that only those services which have to be made public are visible.

Besides the notion of components, a scripting language must offer a set of mechanisms which allow one to connect provided and required services of corresponding components. Such mechanisms can be as "low-level" as a function call or as higher-level as the pipe-operator in the Bourne Shell language. One can view such mechanisms as abstractions over components:[4] the connection of two (or more services) results in a "new" component where the corresponding services are connected.

An important aspect about encapsulation and wiring is whether a language is *compositionally complete* (i.e. it is possible to encapsulate a composition of components as a composite component). For example, a Bourne Shell script can be used as a component of another shell script whereas to our knowledge it was not possible to define ActiveX components in Visual Basic prior to version 5.0 [Mic97].

Foreign code concept. In order to use components not written in a scripting language itself, it is necessary that such a language has features to interoperate with components written in foreign languages. We denote such features as foreign code concept, which is also known as *external extensibility*.

Such a feature is important if a component is implemented as a composition of other components, but does not completely fulfill all of its requirements (e.g. it does not have the required run-time performance). In such a case, it is possible to reimplement this component in an other programming language with more favourable run-time behaviour and integrate this new component into the scripting language using interoperability features.

There are several possibilities how a foreign code concept can be defined for a language: based on i) a programming language, ii) a language-neutral interface description language, iii) binary standard, and iv) operating system resources.

A widely used foreign code concept is based on a programming language: a scripting language specifies interfaces for defining abstractions in an other language. It is very popular (in particular for those languages developed in a UNIX environment) to offer an interface for C or C++. It is often the case that the run-time system of a language offering a C/C++ interface is implemented using C or C++.

A second approach is based on a language-neutral interface description language (often referred to as an IDL), which allows developers to define interfaces in a language-independent format. Such an approach has the advantage that components can be implemented in any language that defines a binding for the interface description language. A well-known example of such an IDL is the IDL of CORBA [OMG96]. From our point of view, the foreign code concept of the Bourne Shell also belongs to this category: the "IDL" defines that commands must be able to read from the standard input stream and produce output onto the standard output and/or error streams, but does not specify any further restrictions on how commands have to be implemented.

Another approach is to define a foreign code concept based on a binary standard. As an

[4] In this setting, the term *abstraction* should be used in functional way.

example, Visual Basic allows developers to directly use ActiveX components, which are based on the *Common Object Model* (COM) [Rog97]. COM specifies a client/component interface at a binary level, independent of any particular programming language or compiler. The interoperability with COM components is probably one of the main reasons why Visual Basic is a very popular scripting language on Windows platforms.

Finally, some scripting languages offer features to interoperate with foreign components using operating system resources (e.g. any program which understands Apple events can be used in AppleScript [Com93]).

One should note that the four possibilities mentioned above do not exclude each other: PythonWin, an implementation of Python [vR96] for the Windows platform, offers extension interfaces for both C/C++ and COM.

An interesting characteristics of a foreign code concept is whether the communication between the "host" language and the foreign components are uni- or bidirectional (i.e. a foreign component can call back into the host language).

Although one might argue that the essential features we have discussed above do not restrict the language space enough, we can justify this list of features as i) they can be immediately deduced from the main purpose of scripting, ii) to our knowledge, all so-called scripting language support these two features, and iii) it is not possible to add other features to the list without excluding some important languages from being scripting languages.

4.2.2 Characterizing features

Besides the essential features we have discussed in the previous section, there are features which are present in many scripting languages. However, from our point of view, they are not essential for scripting itself, but classify scripting languages in the design space. The following list also contains features which are not only important for scripting languages in particular, but for any kind of programming language in general.

- **Embeddability:** some scripting languages are either directly embedded into an application or component (e.g. JavaScript [Fla97] is only available in a web browser), others offer an interface to embed them into other programming languages (e.g. Python offers an interface for C/C++).

- **Extensibility:** scripting languages often offer the possibility to extend themselves with additional abstractions (new components and connectors). As an example, the core of Tcl [Ous94] does not offer the concept of classes and objects, but the stooops extension (which is fully written in Tcl) introduces abstractions for object-oriented programming [Fon98].

- **Objects:** a comparison of popular scripting languages reveals that they either directly support the notion of objects or there are extensions which introduce objects.

- **Exceptions:** for programming in the large and for testing applications, it is useful if a language has features to explicitly cope with errors and exceptions.

- **Execution model:** a criterion to distinguish scripting languages is whether they are event driven or data driven. In the case of an event driven language, it is important to know what kind of *call-back* mechanisms it supports and how *closures*[5] can be specified.

- **Concurrency:** Some scripting languages are inherently concurrent (e.g. the commands of the Bourne Shell), other have extensions which introduce concurrency features (e.g. threads, monitors). In both cases, the kind of higher-level *coordination abstractions* are of interest.

- **Introspection:** scripting languages generally offer features for run-time introspection or even reflection, although these features often only have a restricted functionality. From our point of view, both dynamic creation and execution of code (often referred to as an *eval*-feature[6]) and the concept of *call-by-name* are part of this dimension. Whereas languages like Tcl only offer low-level introspection mechanisms, languages like Python go a step further and offer meta-level protocols.

- **Typing:** According to Ousterhout [Ous98], scripting languages tend to be weakly typed. However, an analysis of popular languages reveals differences in the type system: some languages are untyped (e.g. Bourne Shell) or dynamically typed (e.g. Perl) whereas others have a mixture of static and dynamic typing (e.g. Visual Basic). This analysis also revealed different strategies for resolving type mismatches (e.g. implicit type conversions vs. exceptions).

- **Scoping rules:** The scope of a name (variable, function etc.) is the range of program instructions over which the name is known. The scoping rules of a language defines the strategy how name-value bindings are established. Most scripting languages tend to by *dynamically scoped*, although there are languages which also offer *static scoping* (refer to section 4.4.1 for more details).

- **Built-in data abstractions:** Besides low-level data abstractions such as integers and strings, many scripting languages offer built-in high-level data abstractions. Examples of such abstractions are key-based data abstractions (e.g. dictionaries), ordered data abstractions (e.g. lists), or data abstractions without a particular order or access strategy (e.g. sets). Besides the data abstractions themselves, many languages offer specialized operations on these data abstractions (such as iterations), and Perl even has a special syntax for these operations.

- **Persistence:** Only few scripting languages offer general-purpose support for making complex configurations or properties of applications and components persistent.

[5]Closures are a concept found in functional programming languages and denote the environment in which an abstraction is executed. They formalize the notion of freezing free variables in lambda expressions [Set89].

[6]For a discussion about the *eval*-feature, refer to section 4.4.

4.2.3 Related aspects

The features we have discussed in the two previous sections are certainly not the only ones which can be used to classify scripting languages. Although the aspects mentioned below are important from other points of view, they are not used as a criteria for our classification. Nevertheless, for reasons of completeness, we list features which appear in other approaches for comparing scripting languages.

- A scripting language is either general purpose (e.g. Python, Tcl) or restricted to the architecture of a particular component framework (e.g. Bourne Shell, AppleScript). In the latter case, the scripting language should offer some syntactic sugar in order to reflect the components and connectors of the component framework in the source code.

- Portability is an important aspect in the discussion about reuse. It is certainly true that there exist many scripting languages which allow a user to port their source to different platforms or environments (e.g. Perl runs on most popular platforms), but there are scripting languages which only work in a well-defined environment (e.g. JavaScript generally only runs in the Netscape browser) or operating system (e.g. AppleScript is only available for MacOS).

- For most scripting languages, performance and resource consumption are expected to be dominated by the components (and not the scripts) and, therefore, the performance of the scripting engine usually is not a major issue. However, it is not necessarily the case that applications written with scripting languages are slow. Perl, for example, is highly optimized for text processing, and it is not an easy task to write a C program with the same functionality which runs faster than a corresponding Perl application.

- According to the definition of van Rossum (see section 4.1), a scripting language should have text processing primitives. The presence of such features heavily depends on the application domain of a language. Whereas Perl or sed [McM78] have been designed for text processing as the main application domain, AppleScript was designed for automating MacOS applications, and does not have any built-in text processing primitives. From our point of view, text processing should not be part of the language itself, but supported by library components.

- The same as for the text processing primitives also applies for regular expressions and support for binary data, which should again be supported by library components.

- The definitions of Welch and van Rossum (see section 4.1) claim that a scripting language should i) be interpreted, and not compiled and ii) offer some form of automatic memory management. One of the reasons why interpretation is claimed to be essential for scripting is the presence of an eval-feature in some languages, and it is obvious that a programming language that offers an eval-feature must be either interpreted or dynamically compiled. However, we argue that both properties cannot be directly deduced from the purpose of scripting (i.e. connecting components) and should, therefore, be considered as general programming language design issue.

4.3 Systems and languages

In this section, we will illustrate the required and characterizing features of scripting languages introduced in the previous section and present a selection of languages (including a sample application for each language) that fulfill the given definition. These languages also illustrate the main dimensions of scripting languages and support specific features which are important in the context of a general-purpose composition language.

4.3.1 Bourne Shell

The Bourne Shell is an interpreted scripting language for the UNIX operating system [Bou78]. In fact, it can be considered as being both a shell and a scripting language and is available on most UNIX platforms.

It offers a simple component model based on *commands* and character streams[7] which can be connected using the pipe operator '|' and file/stream redirectors ('<', '>' etc.). The Bourne Shell does not make any syntactical difference between built-in commands (e.g. `read`, `write`, `trap`, and `exec`) and external commands (such as `grep` or `comm`). Commands can be implemented in any programming language, provided they support the ability to read from the standard input stream and produce output onto the standard output and/or error streams. In addition, it is useful if a language offers abstractions for reading command line arguments. Any command which fulfills the properties mentioned above can be used in Bourne Shell scripts. In particular, if a composition of commands is stored as a shell script, this script can be used as a command in other scripts. Therefore, the Bourne Shell is compositionally complete.

Commands which are connected by a pipe operator may execute concurrently (this depends on the actual implementation of the pipe operator). Therefore, the pipe operator does not only connect two commands with a character stream, but also works as a synchronizer (or coordinator) between commands. Furthermore, Bourne Shell scripts are often associated with a *pipe and filter* architectural style, as many scripts have such an architecture. However, scripts are not restricted to this style, and it is possible to define more complex unidirectional data-flow architectures by connecting the standard error stream of a command to the standard input stream of another command using the operator '|&'.

Strings are the only data-type (i.e. variables can only hold string values) and there is a clear separation between the name of a variable (e.g. `var`) and the value it holds, which can be accessed using `$var` (this is often referred to as *variable substitution*). However, there are no sophisticated concepts for data encapsulation: all variables and commands

[7] A text file can be considered as a persistent character stream.

```
awk '! /^#/ {print $1}' keywords | \  # get first word of non '#' lines
    sort > /tmp/$$                    # sort into temporary file

wrong="`grep -h '^#Keys' $* | \       # get lines with '#Keys' tag
    tr -c '[A-Z][a-z]' '[\012*]' | \  # split words into separate lines
    grep -v 'Keys' | \                # remove lines with '#Keys' tag
    sort -u | \                       # sort, remove duplicates
    comm -13 /tmp/$$ - `"             # compare with temporary file

if [ -n "$wrong" ] ; then             # empty string in '$wrong'?
    echo "There are unknown keywords:"
    for i in $wrong ; do              # iterate over unknown keywords
        grep -n "^#Keys:.*$i" *       # display files and line numbers
    done                              # of unknown keywords
else
    echo "All keywords are known"
fi

rm /tmp/$$                            # remove temporary file
```

Figure 4.2: Extracting keywords in the Bourne Shell.

are in a global scope. A string denoting a numerical expression can be evaluated using the command `expr` whereas strings consisting of a sequence of commands can be evaluated using the command `eval`. It is important to note that the usage of `eval` also causes a second round of variable substitutions.

The Bourne Shell offers a restricted set of built-in control structures (such as conditionals and loops) and it is possible to define functions and procedures. Furthermore, each Bourne Shell script has a set of predefined variables which can be used to access the command line arguments passed to the script : `$0` is the name of the command itself whereas `$1` up to `$9` are the values of the first nine command line arguments. In addition, all command line arguments (except `$0`) can be accessed using `$*`. The same set of variables can also be used to access arguments passed to user-defined functions. Due to the fact that the name of a script is available in the script itself, it is possible to define recursive call structures at a script level.

As an example of how the Bourne Shell can be used as a scripting language, consider the keyword extraction script in Figure 4.2. In the context of software architectures in section 3.1, we have discussed a simplified version of this script where the list of keywords is in a format suitable for the `comm` command. However, the script illustrated in Figure 4.2 assumes that the keywords are stored in a text file which contains additional information about each keyword. More precisely, a keyword is the first word of a line not starting with a hash character. Therefore, the actual keywords have to be extracted from this text file before they can be used for comparison.

4.3.2 Tcl

Tcl (as an acronym for *Tool Command Language*) is a dynamically compiled, string-based command and scripting language which is available on most popular platforms [Ous94].

The basic abstraction in Tcl is the *command*, comparable with the concept of a procedure in imperative programming languages. A command has a name (i.e. a string representation), accepts a list of arguments, and executes a sequence of other (possibly built-in) commands. Tcl, like the Bourne Shell, does not make the distinction between built-in commands and user-defined (or library) commands. However, Tcl goes a step further: every programming constructs is achieved with a command, not syntax; even the basic control structures like conditionals and iterations are implemented as commands. Of course, some of these commands are in bed with the interpreter,[8] but in principle it is possible to remove, replace, or add new commands that provide "traditional" programming features. The only commands which must be supported by a Tcl interpreter and cannot be defined in terms of other commands are `proc` (which enables the definition of new commands) and `set` (for updating and accessing variables).

The syntax of Tcl is only about grouping arguments and substituting values. The first string found on a new line is considered to be the name of a command whereas the rest of the line are the arguments for this command. The only special syntax is for comments (lines starting with a hash symbol), for variable substitution (preceding a variable name with a $ sign), and for grouping expressions. Expressions can be grouped using curly braces (which prevents substitutions and considers the expression as a single string), double-quotes (which allows variable substitutions), and brackets (which results in both variable and command substitutions). The semantics of a Tcl script comes from the run-time behaviour of each command and can in generally not be deduced from the program text alone.

Similar to the Bourne Shell, strings are the only basic data-type. Although there are higher-level data-types (like lists and arrays), they can all be encoded as strings.[9] Tcl is dynamically scoped (i.e. the value of a variable is defined in terms of program execution), and each command defines its own (private) scope. In order to access variables from an outer scope, they must either be declared as global variables or accessed using the command `upvar`. Since global variables were the only means of making information persistent from one call of a command to the next, the latest version of Tcl introduced the notion of *namespaces* which enables the declaration of variables (and commands) local to a namespace.

In contrast to the Bourne Shell, Tcl offers some restricted introspection and reflection facilities. Using the command `info` with appropriate arguments, it is possible to explore the internal state of the run-time system. The command `eval` can be used to execute dynamically created code. A similar construct is the command `uplevel` which enables the execution of a sequence of commands in an outer scope.

[8]Although the run-time system of the lastest versions of Tcl use dynamic compilation instead of pure interpretation, it is still referred to as an interpreter.

[9]For efficiency reasons, higher-level data abstractions and the corresponding commands for update etc. are built-in in the latest versions of the run-time system.

```
set KeyWordFile "keywords"
set fileId [open $KeyWordFile r]
foreach line [split [read $fileId] \n] {
    if [regexp {^([A-Za-z\+]+).*} $line match first] {
        lappend KeyWords $first
    }
}
close $fileId

foreach file $argv {
    set fileId [open $file r]
    foreach line [split [read $fileId] \n] {
        if [regexp {^(#Keys:)  (.*)} $line match tag words] {
            foreach word $words {
                if {[lsearch -exact $KeyWords $word] == -1} {
                    puts stdout [join [list $file $line] ":  "]
                }
            }
        }
    }
    close $fileId
}
```

Figure 4.3: Extracting keywords in Tcl.

One major reason why Tcl became very popular is the Tk extension for building graphical user-interfaces. Tk extends Tcl with a set of commands that create and manipulate so-called *widgets*. Widgets are windows in graphical user interfaces and have a particular behaviour and appearance. Tk offers a set of commands which allow a user to define simple, but powerful interconnections between widgets. Tk was the first high-level tool for implementing portable graphical user interfaces and has been incorporated as a library into many other scripting languages (e.g. Perl and Python).

As an example to show the way Tcl can be used for scripting, we have implemented the keyword extraction script mentioned in the previous section in Tcl as well (refer to Figure 4.3). In contrast to the Bourne Shell script which is declarative, Tcl adopts an imperative style of programming.

Tcl is a good example of an extensible scripting language: it allows the definition of user-defined abstractions (i.e. new commands) sharing the same syntactical framework as is used for built-in commands. This is possible because of the sheer simplicity of Tcl's underlying design: the Tcl language consists of an interpreter with a small set of rules for parsing arguments and performing variable and command substitutions, respectively. Because Tcl commands are viewed merely as lists of arguments, the first of which is the command name, users are permitted to define their own constructions in the same manner that Tcl itself defines built-in commands. As an example, TclBlend [Joh98] is a library of Tcl commands which allows the Tcl run-time system to interoperate with a Java Virtual

Machine. For each Java object instantiated in the virtual machine, the Tcl run-time system creates a new command as a proxy. An invocation of such a command interprets the first argument as the method selector and forwards the method call to the corresponding Java object (using the Java Native Interface JNI [Sun97a]). Therefore, Tcl is an important step into the direction of a composition language where all abstractions can be implemented as library extensions.

4.3.3 Perl

Perl (as an acronym for *Practical Extraction and Report Language*) is a dynamically compiled (object-oriented) scripting language and can be considered as a uniform selected merge of several other (scripting and shell) languages such as sed, awk, csh, and C [WCS96].

As mentioned in the acronym, one main application area of Perl is extracting and reformatting information of (text-)files. Therefore, there are many built-in (and often highly optimized) abstractions for text and stream processing, pattern matching, text substitution etc. This is one of the reasons why Perl is often associated with CGI programming since these built-in abstractions turn out to be particularly suitable for extracting information out of HTML-forms and generating new HTML pages.

In contrast to the Bourne Shell and Tcl, Perl does not only offer built-in support for scalars (integers, strings, booleans), but also for higher-level data abstractions such as lists, arrays, and hashes (associative arrays). The first character of a variable defines the type of its contents (e.g. a variable starting with a $ sign holds scalar values whereas variables starting with a @ sign holds array values). A variable of one type cannot hold a value of another type, and assignment to a variable of another type causes data coercion. In addition, each variable type has its own namespace (i.e. a scalar variable `$foo` does not override an array `@foo`). Several operators apply built-in coercions for evaluation expressions. As an example, the `'+'` operator is used to add integers. If one of the operands is a string containing an integer value, the string is converted into an integer and the summation is performed with integers.

Perl introduces the notion of *contexts* for evaluating expressions. There are two main contexts: *scalar* and *list*. The scalar context can be further classified into *string* context, *numeric* context, and *don't care* context. The evaluation of expressions heavily depends on the requirements of the context where it is evaluated in. For example, assignment to a scalar variable evaluates the right-hand side in a scalar context whereas an assignment to an array or hash evaluates the right-hand side in a list context. Certain built-in operators behave differently depending on the context of their result. More precisely, these operators are overloaded on the type of their return value. One should note that this is the only way Perl offers (limited) support for operator overloading.

Perl allows the definition of new functions and procedures, which are referred to as *subroutines*. It is possible to define both named and anonymous subroutines. However, only the latter are first-class values: named subroutines cannot be assigned to a variable, passed as an argument to another subroutine, or returned as a result. Subroutine definitions can be nested, may return multiple values, and are defined without explicit formal parameters. The actual parameters can be accessed using the variable `@_`. It is good practice to assign `@_` to a list of locally declared variables in order to emulate formal parameters. This concept of parameter passing implies that it is not possible to define default values for arguments. It makes a difference whether the local variables emulating the formal parameters are declared using `local` or `my`: the former causes *dynamical name lookup* whereas the latter causes *lexical name lookup*. This is of importance if a "formal" parameter is used as a free variable in a nested subroutine definition.[10] Parameters to subroutines are passed by value (an not by reference), unless they are explicitly passed as a reference type (see below).

In contrast to Tcl, Perl makes an explicit difference between built-in commands and user-defined commands (i.e. subroutines): for invocation, the latter have to be preceded with an `&`. Many built-in commands have implicit default parameters (e.g. `$_` in `split(/ /)`, which causes a (line-)string to be split into separate words), which allows users to write very compact source code, but introduces certain problems for readability and maintainability.

The latest version of Perl (version 5) adds abstractions for classes, objects, references, and hashes (i.e. associative arrays). These new abstractions do not integrate smoothly with existing abstractions and code written for Perl 4 might break.

Perl offers an extension interface for both C and C++. However, in contrast to other scripting languages, Perl is generally not embedded, although an embedding interface for C and C++ is available.

As an example of a Perl script, consider the implementation of the keyword extraction script mentioned previously in this section (see Figure 4.4). Although the Perl implementation is less verbose than the corresponding Tcl script, the structure is very similar. However, instead of using a list as the data-structure for storing the keywords, we have used the built-in hash abstraction in order to emulate a set.

4.3.4 Python

Python is an object-oriented scripting language that supports both scripting and programming in the large [vR96]. It supports objects, classes, single and multiple inheritance, modules as well as a run-time (meta-)object protocol[11].

Similar to Tcl, Python has an unifying concept: everything is an object, including functions and classes.[12] Objects are first-class values and, therefore, any abstraction can be used as a first-class value.

Besides the user-defined methods, each object (including class objects) has a number of methods which Python assigns a special interpretation. The names of these methods begin and end with two underscores in order to distinguish them from library or user-

[10] The different behaviour of `my` and `local` is in general not explained correctly in most Perl books.

[11] In the Python community, this protocol is often referred to as the *Python Object Protocol*.

[12] This quote is not fully true as *built-in* functions such as `print` have a slightly different behaviour, but it applies to all library or user-defined abstractions.

```
$KeyWordFile = "keywords";          # assignment to scalar variable

open (IN, "<" . $KeyWordFile) || die "Can't open $KeyWordFile";
while (<IN>){                       # loop over all lines of IN
    if (/^(\w+)/) {                 # search for word at line start
        $KeyWords{$1} = 1;          # assign first word to hash table
    }
}
close (IN);                         # close keyword file

while (<>){                         # loop over stdin or cmd line args
    chop;                           # remove last character of line
    if (/^#Keys:/){
        ($key, @line) = split (/ /); # split line, assign rest to array
        foreach $word (@line) {     # loop over all words of in array
            $KeyWords{$word} || print $ARGV . ": ". $_ ;
        }                           # print word if not in keywords
    }
}
```

Figure 4.4: Extracting keywords in Perl.

defined methods. Some methods implement basic type operations (getting the value of an index of a collection `x[i]` results in calling `x.__getitem__(i)`), some overload expression operators (e.g. `a + b` is interpreted as `a.__add__(b)`), and others intercept class behaviour (see below).

The name lookup of methods is based on a metaobject feature: each object has a method `__getattr__` which is called whenever the default mechanism of attribute lookup (e.g. a method call) fails. As an example, the method call `x.foo (3)` is translated into `apply (getattr(x,"foo"),(x,3))`. In particular, the built-in function `getattr` is called which *dynamically* looks up the method `foo` in the method dictionary of the class object of `x`. If `foo` is not defined there, then the superclass objects are searched for this method in a depth-first, left-to-right order. If the method is not found in any of the classes, the method `__getattr__` is invoked. Since this method can be overridden in any class, Python defines a powerful hook for meta-programming [Lut96] (refer to section 8.8 for further details about the (meta-)object protocol).

Similar to C++, a function is an object which implements the method `__call__`. Python allows three different ways of defining functions: i) as instances of a class which implements a method `__call__`, ii) using the keyword `def` which defines named functions, and iii) using the `lambda` construct which defines anonymous (i.e. unnamed) functions. The last form of function definition is particularly useful since it is often needed to pass a function as a parameter to another function. Functions allow default values for parameters and argument passing either by position or name.

Python has a so-called 3-scope namespace lookup rule to resolve name references: it defines i) a local namespace which can be thought of as a dictionary that contains all locally defined names, ii) a global namespace which contains all globally defined names, and iii) a built-in namespace containing all built-in names. Due to the 3-scope namespace lookup rule, namespaces are not nested, and an abstraction has no access to the local names of its enclosing scope (unless this happens to be the global scope). Furthermore, Python is dynamically scoped and, therefore, all names are dynamically resolved using the 3-scope namespace lookup rule, including free names in lambda expressions.

Besides integers and strings, Python has a number of built-in higher-level data abstractions such as big numbers, lists, arrays, and dictionaries. In addition, by using features of the (meta-)object protocol, it is easily possible to add new collection abstractions such as sets and streams. In contrast to Perl, Python does not perform any implicit data coercions (e.g. `"2" + 3` leads to a type error) and variables can hold values of any type during their lifetime.

Another important feature of Python is the support for *modules*. The source of a script file consists of *imports* (which specifies which modules are imported by a script file) and a sequence of *declarations* (classes, functions, constants etc.). By testing the special attribute `__name__`, it is possible to check whether the contents of a script file is executed as the main or top-level script or if it is imported into another script file. In the former case, all declarations get executed whereas in the latter case, only some of the declarations get imported into the enclosing script file.[13] Furthermore, it is possible to add initialization code if this is required for a correct import of a module. The run-time system keeps track of all imported modules and ensures that each module only gets imported once. Equivalent to the `CLASSPATH` environment variable of the Java run-time environment, the Python run-time system uses the `PYTHON_PATH` variable to search for the source files of imported modules.

Python is available on most popular platforms, and there exists both an implementation in C/C++ and Java. The former has an extension and embedding interface for C and C++ whereas the latter supports Java extensions and embeddings. The Windows implementation has an additional interface to COM/ActiveX, and on most UNIX platforms, a CORBA binding is defined.

As an example of an application in Python, consider the keyword extraction script in Figure 4.5. This script uses two non-standard Python libraries `set` which implements a simple set abstraction and `miniStream` which defines a framework for streams, filters, and transformers. For further information about the abstractions used in this example, refer to section 5.1 or appendix A.

4.3.5 AppleScript

AppleScript is a dynamically typed, event- and object-oriented scripting language which only runs on the MacOS platform. In fact, AppleScript is not a scripting language on

[13] Python supports two different import statements: i) `import aModule` imports all declarations of the file `aModule.py` into the local scope of the import statement, and ii) `from aModule import` *nameList* only imports the declarations specified in *nameList*.

```
import re, string, sys              # import external modules
from set import *
from miniStream import *

KeyWordFile = "keywords"
re1 = re.compile ("^[A-Za-z\+]+.*$") # search pattern keywords
def getFirstWord (line):            # get first word of a line
    return string.split (line)[0]

KeyWords = Set (FileStream (open (KeyWordFile, 'r'))
                | Filter (lambda line:  (re1.match (line) > 0))
                | Transform (getFirstWord)
                )

keyline = re.compile ("^#Keys:.*$")  # search pattern for keyword lines
def checkLine (line):               # check for unknown keywords
    for word in string.split (line)[1:]:  # split line into words
        if not word in KeyWords:  return 1
    return 0

def transform (line, fileName):     # function for transformation
    return string.joinfields ([fileName, string.strip (line)], ":  ")
for arg in sys.argv[1:]:            # loop over arguments
    for line in (FileStream (open (arg, 'r'))
                 | Filter (lambda line:  (keyline.match (line) > 0))
                 | Filter (checkLine)
                 | Transform (lambda line:  transform (line, arg))
                 ):
        print line
```

Figure 4.5: Extracting keywords in Python.

its own: it is a front-end to the *Open Scripting Architecture* defined for OpenDoc in the context of the *System Object Model* (SOM) [FM96].

The Open Scripting Architecture (OSA) of OpenDoc is an enabling technology that views applications as collections of *component parts*. These parts can be tied together via scripts written in a variety of scripting languages. OSA defines how containers and parts communicate with each other using an event registry, which contains list of commands that evoke responses from parts.

Semantic events are used to manipulate the content model of a part: it is not only possible to act on parts, but also on content objects within parts (e.g. the second word of the first paragraph of a document). Semantic events are the means by which scripting systems manipulate the contents of a scriptable part. For a part to be scriptable, its part editor must surface the list of operations on its content objects that can be invoked via semantic events.

Every scripting language that supports open scripting must be able to translate elements of the language into semantic events. In addition, it must provide an interface that lets OpenDoc parts manipulate scripts, record them, and attach them to events.

The Open Scripting Architecture defines four levels of scriptability:

- *scriptability*: a part editor exports all supported operations and can accept semantic events,
- *customizability*: scripts (i.e. new behaviour) can be attached to visible interface elements (e.g. buttons),
- *tinkerabiliy*: attach new scripts to events received or generated by a part. Tinkerability is the union of scriptability and customizability.
- *recordability*: record all events received or generated by a part.

As mentioned above, AppleScript is a scripting language which implements a front-end to the Open Scripting Architecture.[14] The main purpose of AppleScript is to automate, integrate, and customize *scriptable* applications. The language supports classes, objects, and inheritance as well as commands and makes the difference between *script objects* (i.e. objects defined in AppleScript using object-oriented abstractions), *part objects* (i.e. component parts in terms of OpenDoc), and *application objects* (objects belonging to part objects).

AppleScript works by sending messages (i.e. Apple events) to applications. Running a script is done by sending statements to the AppleScript extension, which interprets the statements and sends Apple events to the appropriate part objects. A command is a series of words used in AppleScript statements to request an action. Every command is directed to a target, which is the object that responds to the command. The target is usually an application object, but it can also be a script object, a user-defined subroutine, or a user-defined value in the current script. Application objects belong to an application such as words or paragraphs in a text document, and every object belongs to a class.

AppleScript defines the notion of *script properties* which denote labeled containers for values which persist until a script is recompiled (see below). Therefore, unlike many other scripting languages, AppleScript has a limited built-in support for persistent data.

Scriptable applications (i.e. component parts) can in general be implemented in any programming language. The only requirement is that they support an appropriate interface for OSA. In particular, AppleScript scripts can be implemented in a way that they support such an interface: they must define handlers for some predefined Apple events (e.g. `Open`, `Run`, and `Quit`). Therefore, AppleScript can be considered as compositionally complete.

In contrast to the other scripting languages we have discussed in this section, AppleScript does not have an equivalent to an "eval" or "exec" feature.

[14] To our knowledge, there exists a Tcl extension for the MacOS implementation which also defines a front-end to OSA.

```
set appName to "FrameMaker 5.5 PowerPC"
set FolderName to (choose folder with prompt
    "Select a folder to open:  ") as string

set FolderListing to list folder (file FolderName)
tell application appName
    repeat with i from 1 to the number of items of FolderListing
        set FolderItem to item i of FolderListing
        if folder of (info for file (FolderName & FolderItem)) is false then
            if file type of (info for file (FolderName & FolderItem))
                is "FASL" then
                open file (FolderName & FolderItem)
                save document 1 in file (FolderName & FolderItem & ".mif")
                    as "MIF "
                close document 1
            end if
        end if
    end repeat
    quit application appName
end tell
```

Figure 4.6: Converting FrameMaker files into MIF with AppleScript.

AppleScript comes with an application called *Script Editor* which can be used to create and modify scripts. The Script Editor also includes an interpreter which enables a direct execution of AppleScript scripts within the environment. In addition, the Script Editor has an integrated compiler and compiled scripts can be executed outside the environment.

As an example of an AppleScript script, consider the script in Figure 4.6. The purpose of this script is to convert all FrameMaker files found in a folder into a MIF (Maker Interchange Format) format, and is a good example how AppleScript can be used to automate tasks. This script is part of the FrameMaker distribution for MacOS.

4.3.6 Manifold

A new class of formalisms has recently evolved for describing concurrent and distributed computations based on the concept of *coordination*. The purpose of a coordination model is similar to the one of software architectures: making a clear separation between computational elements and their relationships. Coordination languages generally provide abstractions for controlling synchronization, communication, creation, and termination of concurrent and distributed computational activities [Arb96]. One can also consider coordination as *scripting of concurrent and distributed components*.

Manifold is a coordination language for managing complex, dynamically changing interconnections among sets of independent, concurrent, and cooperating processes [Arb98]. Like most so-called *process-oriented* coordination languages, Manifold clearly distinguishes between *computational processes* (written in any conventional programming language) which can be augmented with some *communication primitives*.

The basic concepts of Manifold are *processes*, *events*, *ports*, and *streams*. A process is an independent, autonomous, active entity and has its own private processor and memory. Manifold processes communicate by means of input/output ports, connected between themselves by means of streams. A stream represents a flow of a sequence of units (of information) between two ports. The evolution of a program (or a *coordination topology* in terms of Manifold) is event-driven based on state transitions. More precisely, a Manifold process is at any moment in time in a certain state where it has set up a network of coordinated processes. When a process observes the raising of some event, it may break off the stream connections and evolve to some other (predefined) state with a different network of coordinated processes. Unlike other process-oriented coordination languages, the events of Manifold are not parameterized and cannot be used to carry data: they are only used for triggering state changes and, therefore, for evolving the network of coordinated processes.

Like in architectural description languages, the conceptual model behind Manifold is based on the separation of computation and communication concerns into different program modules. However, this separation of concerns goes a step further and it is possible to abstract specific computation and communication concerns into reusable modules.

Although Manifold does not directly provide any primitives for computation, it is a computationally complete language since computation can be built on top of the built-in communication primitives. As such it advocates the point of view that all computations can be expressed as interactions, similar to the concept of communicating names in the π-calculus.

Due to the fact that computation and coordinator processes are indistinguishable from the point of view of other processes, coordinator processes can, recursively, manage the communication of other coordinator processes, just as if they were computation processes themselves. This implies that any coordinator can be used as a higher-level or meta-coordinator in order to build reusable, higher-level coordination protocols.

As an example of a Manifold program, consider the bucket sorter described in Figure 4.7. The process `Sorter` initially activates a computation process `AtomicSorter` which performs the actual sorting. This process is able to sort a bucket with at most `k` units (i.e. elements to be sorted) and will raise an event `filled` once it has received the maximum number of units to sort. When the process `Sorter` detects this event, it activates i) a new sorter process and ii) a merger process which is responsible for merging the output of both sorters into a single stream. Depending on the bucket size `k` and the number of units to be sorted, an arbitrary number of sorter and merger processes may be created and linked together at run-time. One should note that any process involved in this example has (by default) an `input` and `output` port.

```
export manifold Sorter () {
    event filled, flushed, finished.
    process atomsort is AtomicSorter(filled).
    stream reconnect KB input input -> *.
    priority filled < finished.

    begin:
       (  activate(atomsort), input -> atomsort,
            guard(input,a_everdisconnected!empty,finished)
       ).

    finished:
       {  igonore filled.
          begin:  atomsort -> output
       }.

    filled:
       {  process merge<a,b | output> is AtomicIntMerger.
          stream KK * -> (merge.a, merge.b).
          stream KK merge -> output.

          begin:
             (  activate(merge),
                input -> Sorter -> merge.a,
                atomsort -> merge.b
                merge -> output
             ).

          end | finished:  .
       }.

    end:
       {  begin:
             (  guard(output,a_disconnected,flushed),
                terminated(void)
             ).

          flushed:  halt.
       }.
}
```

Figure 4.7: Bucket sorter in Manifold.

4.3.7 Summary of concepts

In the following, we briefly summarize the important concepts and features we identified in the scripting languages discussed in this section.

- The **Bourne Shell** is an interpreted scripting language for the UNIX operating system and offers a simple component model based on commands and character streams. Commands can be connected by using higher-level connectors (e.g. the pipe operator '|'), which make the architecture of a Bourne Shell script explicit in the source code. It is compositionally complete (a composition of commands is again a command) and supports a declarative style of programming.

- **Tcl** is a dynamically compiled, string-based scripting language and is available on all popular platforms. The basic abstraction in Tcl is a command, and since every programming construct is achieved with commands (and not special syntax), commands are the unifying concept of the language. The concept of commands allows a user to extend the language using the same syntactical framework as is used for all built-in commands.

- **Perl** can be considered as a uniform selected merge of sed, awk, csh, and C. It offers higher-level data abstractions such as lists, arrays, and hashes and syntactic sugar for processing instances of these higher-level data abstractions. Perl introduces the notion of contexts for evaluating expressions, offers support for operator overloading based on contexts, and has both lexical and dynamic scoping rules.

- **Python** is an object-oriented scripting languages that supports both scripting and programming in the large. Objects are the unifying concept (i.e. "everything is an object") and, therefore, all abstractions are first-class values. Python offers a meta-level protocol which can be used for extending and adapting existing abstractions as well as for operator overloading. Finally, the language model supports keyword-based parameter passing.

- **AppleScript** is a dynamically typed, event- and object-oriented scripting language which only runs on the MacOS platform. It is a front-end to the Open Scripting Architecture and offers a component model based on scriptable applications (also known as component parts). The main purpose is to automate, integrate, and customize scriptable applications. It is compositionally complete, but in contrast to many other scripting languages, it does not offer an equivalent to an "eval" feature.

- **Manifold** is a coordination language for managing complex, dynamically changing interconnections among sets of independent, concurrent, and cooperating processes, and should be considered as a scripting language for concurrent and distributed components. It is particularly suitable for specifying and implementing reusable, higher-level coordination abstractions and protocols as well as for dynamically evolving architectures.

The reader should note that other scripting languages (e.g. DCL [Ana89], Icon [GG96], JavaScript [Fla97], Lua [IdFCF96], Obliq [Car95], Rexx [Cow90], or Visual Basic [Mic97]) also support some of the features we have illustrate in this section, but a detailed discussion of all of these languages is beyond the scope of this work.

4.4 Dynamic aspects of scripting languages

Several popular scripting languages offer abstractions for executing dynamically created code (also known as an "eval" feature), and it is often pointed out that such a feature is essential for scripting.[15] We have claimed in section 4.2.2 that an "eval" feature is not essential for scripting, but should be considered as a characterizing feature. In this section, we justify this claim by showing that an "eval" feature is useful for implementing adaptable and extensible components, but sketch how these components could be implemented without an "eval" feature. We also illustrate that eval-like features differ from language to language.

The behaviour of eval-like features heavily depends on the scoping rules and namespace concepts of the corresponding languages, and an analysis of the properties cannot be separated from the discussion of these aspects. Therefore, we briefly introduce and discuss the concepts of static and dynamic scoping, namespaces, and closures.

4.4.1 Scoping rules, namespaces

As pointed out in section 4.2.2, the *scoping rules* of a programming language define i) the range of program instructions over which a *name* (a variable, function, type constructor etc.) is known and ii) how name-value bindings are established (i.e. which values for names have to be taken for evaluating expressions or declarations).

Names can be bound to a scope either statically or dynamically. *Static scope binding* (also referred to as *lexical scoping*) defines the scope of a name in terms of the lexical structure of a program: each reference to a name can be statically bound to a particular (implicit or explicit) declaration by examining the program text alone. Static scoping rules are adopted by most compiled programming languages such a C/C++, Java, Eiffel and Visual Basic.

Dynamic scope binding defines the scope of a name in terms of program execution. Typically, each name declaration extends its effect over all the instruction executed thereafter, until a new declaration for the same name is encountered during execution. Examples of programming languages adopting dynamic scoping rules are APL or LISP.

The scoping rules of a programming language also define whether names defined in an enclosing (or outer) scope can be accessed in a enclosed (or inner) scope. This is often referred to as *nested scoping*. Depending on the concrete scoping rules, the "outer" scope

[15] Some authors consider a language a scripting language when it is easily possible to build a run-time environment for the language by only using the language itself. This is rather a trivial task when a language offers an "eval" feature.

has a different meaning: for lexical scoping, the outer scope refers to the enclosing scope where an abstraction is lexically defined. For dynamic scoping, however, the ”outer” scope generally refers to the callers scope (see the discussion of Tcl below). Pascal is an example of a language with static scoping rules where nested scoping is supported.

Programming languages generally support different (often implicit) *namespaces* for different abstractions. For example, C makes the distinction between names for types and names for variables and functions, respectively (i.e. it is possible to define a variable `foo` of type `foo`). Tcl uses the term namespace in different context: a namespace is an abstraction to group names in order to define hierarchies of named scopes.

The main purpose of a namespace concept is to offer support for structuring larger applications, and to avoid the usage of (often error-prone) naming conventions. In addition, it can be seen as an important glue mechanism which enhances the reusability of components written in a language without explicit namespace support, as it is a known practice in these languages to use global variables for exchanging data. Common to most programming languages is that they only offer very restricted possibilities to control, modify, or extend the scoping rules.

4.4.2 Eval

Experiments have shown that eval-like features of scripting languages have a lot in common (e.g. they require a string denoting the code which has to executed as an argument), but their behaviour may differ considerably. In particular, the behaviour depends on the underlying scoping rules and namespace structure, on the actual arguments given, on the way the language implements substitution, and how exceptions within the code to be executed is handled.

In the following, we discuss the properties of eval-like features found in Perl, Python, and Tcl. A detailed comparison of eval-like features found in other scripting languages such as JavaScript or Rebol is beyond the scope of this work.

Perl. Perl supports a simple “eval” feature `eval` which takes either a single expression or a complete block as an argument. The string value representing the expression or block is parsed and executed in the context of the scope where `eval` is invoked. Any variable settings or subroutine definitions are still accessible after the execution has terminated (similar to the execution of a subroutine), although locally scoped variables declared within the eval-block (either using `local` or `my`) last only until `eval` is done. If no argument is given to `eval`, the current contents of the pseudo-variable `$_` is used as argument string. The value returned from an `eval` is the value of the last expression evaluated. If a syntax or run-time error occurs during the evaluation, `eval` returns an undefined value and assigns an error message to the pseudo-variable `$@`. An important fact to know is that the string passed as an argument is parsed and evaluated every time `eval` is invoked and, therefore, references to variables are replaced by the values they hold during execution, and not during definition. This is of particular interested when an `eval` is used in the body of a subroutine.

Since the `eval` of Perl traps most errors,[16] including fatal errors which can not be handled otherwise, `eval` is the most secure way to do all exception handling in Perl. Prior to version 5, Perl did not have any built-in support for references or nested data structures, and using `eval` was the only possibility to emulate these kind of data structures.

Python. In contrast to Perl, Python has two different eval-like features: `eval` runs a string containing a Python *expression* and returns its result whereas `exec` runs a string containing a *statement* and does not return a result. Without additional arguments, both features run the code-string in the scope of the caller (i.e. in the same local, global, and built-in namespaces which are used for name lookups). In particular, the code-string can modify the contents of the local and global namespaces of its caller. Since this is sometimes not desirable, in particular in the context of executing "untrusted" code, Python enables to explicitly pass a global and local namespace to the commands `eval` and `exec`, respectively. This implies that for any name lookup, the namespaces passed as arguments are used and not the ones "accessible" by the caller. Therefore, a complete separation of the two scopes is achieved and the code-string passed to `eval` or `exec` can be executed in a protected environment. In addition, the namespaces passed as arguments can be "primed" with default values in order to achieve a particular behaviour (see the calculator example later in this section). If the dynamic evaluation of code causes an error or raises an exception, the corresponding error or exception is passed to the caller and can be handled appropriately.

Due to the fact that the interpreter included in the Python run-time system can be invoked either using `eval` or `exec`, Python can be used as an embedding language for components written in Python itself.

Tcl. Tcl offers several possibilities to execute dynamically created code: i) using the command `eval` (possibly annotated with an explicit namespace), ii) using the command `uplevel`, and iii) using `eval` in a nested interpreter.

Similar to Perl or Python, Tcl has a command `eval` which takes a command string as an argument and dynamically executes this string. The execution of a command string may cause the modification of the current scope. In addition, by using the command `namespace` (e.g. `namespace eval` *`namespace $cmd`*), it is possible to execute a command string *`$cmd`* in the scope of the explicitly given namespace *`namespace`*.

The command `uplevel` is similar to `eval` as it takes a code-string as an argument which is dynamically executed. However, the code-string in `uplevel` is executed in an outer scope, which is essential for defining new control structures entirely in Tcl.

The problem with the approaches mentioned above is that an `exit` in the command string causes the termination of the whole program. In order to avoid this problem, the latest version of Tcl offers the concept of *nested interpreters*. Any Tcl program can instantiate (a possibly nested hierarchy of) child interpreters which can be used for evaluating Tcl commands. A nested interpreter defines its own namespaces which are separated from

[16]The reader should note that an `exit` cannot be caught using `eval` and causes the program to stop whereas a statement `die` is caught.

the namespaces of the corresponding parent interpreter. In addition, a parent interpreter has the possibility to disallow or redefine certain commands in nested interpreters. In particular, the command `exit` can be redefined in a way that only the nested interpreter terminates upon execution of `exit`, but the corresponding parent interpreter is not affected. The concept of nested interpreters is very similar to the `eval` of Python augmented with explicitly given namespaces.

The reader should note that all the standard grouping and substitution is done on the command string before any of the eval-like features mentioned above are executed. If the string is enclosed with double-quotes, any occurrence of a variable is substituted by the current value whereas enclosing with curly braces prevents variable substitution. In the latter case, variable substitution occurs in the scope where the eval-like feature is executed.

One of the main reasons for using the command `eval` is the call-back mechanism of Tk. Most Tk widgets execute so-called *call-back commands* whenever a specific event occurs (e.g. a button is pressed). These call-back commands can generally be specified using the `-command` option upon instantiation of a widget. The problem is that a call-back command is neither executed in the scope where it was defined nor in the scope it occurs, but in the *global* scope. In order to avoid variable substitution in the global scope (which is probably not what is intended), call-back commands are often defined using `eval` with already substituted variables (i.e. enclosing the command to `eval` with double-quotes).

Example: a simple calculator. In the following, we illustrate the usefulness of an "eval" feature as a string-based interface to an adaptable interpreter component by implementing the simple calculator shown in Figure 4.8. The calculator consists of an *entry* field at the top (to display and change expressions) and six rows of *buttons* for entering numbers, names of variables, and operations. Furthermore, it supports the four basic operations (addition, subtraction, multiplication, and division), grouping of expressions using parenthesis, and assignment to variables. The button eval is used to evaluate the current expression and to display the result in the entry field.

In fact, the calculator is built of two components: i) a component for representing the graphical user interface and ii) a component for evaluating calculations (which we will denote as *evaluator* in the following). The graphical user interface is a composition of finer-grained elements such as buttons and text fields. For the following discussion, the evaluator is of more interest.

The operations of the calculator can be expressed by a context-free grammar. Therefore, the evaluator can be seen as a small interpreter. As a first approach, it is possible to implement the evaluator using standard interpreter technology [ASU86]. Due to the fact that the grammar is rather simple, implementing a small interpreter is not a difficult, but certainly not a trivial task, and has the advantage that we have complete control of how expressions are parsed and evaluated, including appropriate error handling facilities. For a sample implementation of such an interpreter (consisting of app. 300 lines of Python code), refer to [Lut96, chapter 16].

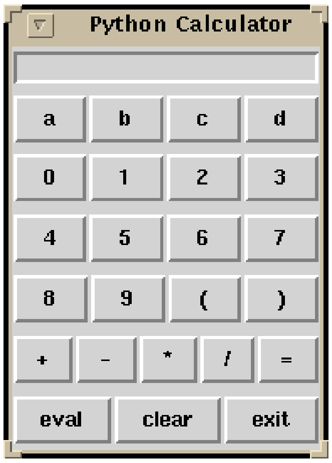


Figure 4.8: A simple calculator GUI.

As noted before, Python's development environment (in particular the interpreter) can always be used in a running Python application. Therefore, as a second approach, we directly use the Python run-time system as the evaluator component: the parsing and evaluation of the expressions is delegated to the built-in abstractions `eval` and `exec`: the former is used to evaluate expressions whereas the latter abstraction is used to evaluate assignments to variables. This approach has the advantage that the implementation of the evaluator consists of considerably less lines of code (see the method `eval` in Figure 4.9) and leads to a more robust system. The reader should note that the evaluator uses i) a private namespace `self._names` for evaluation and ii) nested exception handlers for a correct exception and error handling (the corresponding code is omitted in Figure 4.9). On the other hand, this approach has the disadvantage that the "language" of the calculator has to be a subset of Python.

A major difference between the two approaches is their flexibility for extensions, and we illustrate the corresponding differences by extending the calculator with complex numbers: all occurrences of the character `i` should treated as the imaginary number i. Both the GUI and the evaluator components have to be modified in order to reflect this extension. The extension of the GUI component is a trivial task as the button d of the upper row can be modified so that it appends an i to the entry field.

Python already has a built-in data type for complex numbers which supports the four basic operations we offer in the calculator. However, the string representation of the built-in complex numbers uses the character `j` for the imaginary number i (e.g. `print complex (2,1)` leads to `(2+1j)`). In order to overcome the problem of having two representations for the same concept, we have written a wrapper class `Complex` which wraps the built-in data type and modifies the string representation according to our needs (i.e. it uses the character `i` instead of `j`).

```
class Evaluator ():                # evaluator component
    def __init__ (self):           # constructor (self as first argument)
        self._names = {}           # private namespace for expressions

    def eval (self, expr):         # evaluate expression 'expr'
        try:
                        # evaluate as expression in private namespace
            v = eval (expr, self._names, self._names)
            return (str (v))       # return result
        except SyntaxError:        # catch syntax errors
            try:
                        # execute as statement in private namespace
                exec (expr, self._names, self._names)
            except:
                ...                # process errors
        except:
            ...                    # process other errors
```

Figure 4.9: Evaluator of a simple calculator in Python.

Using the class `Complex`, the extension for complex numbers using the approach with `eval` and `exec` only requires two additional lines of Python code: we simply prime the private namespace `self._names` of the calculator with the class `Complex` and the imaginary number `i`:

```
exec ("import complex", self._names, self._names)
exec ("i = complex.Complex (0,1)", self._names, self._names)
```

Extending the first approach (i.e. the interpreter) is in general more complicated and heavily depends on the data representation used by the parser and how the resulting parse tree is evaluated. However, the evaluator approach using an "eval" feature also has its limitations. For obvious reasons it is not possible to extend the calculator with abstractions that cannot be expressed with Python syntax. Such limitations are generally not a problem if an interpreter approach is used.

One of the main reasons why eval-like features are used in most languages is to emulate abstractions which are not supported by the language itself or to overcome specific drawbacks (e.g. Perl does not allow the usage of variables in pattern matching commands such as `tr`, but embedding such a command in `eval` overcomes this drawback and variables can be used). Abstracting from these typical usages of eval features, we come to the conclusion that any eval-like feature should be considered as a *string-based interface* to a reusable and adaptable *interpreter component*. It can be assumed that the built-in interpreter called by an eval feature is robust, has been extensively tested and, therefore, any application using an eval feature requires less testing than a comparable application which uses a user-defined parser and/or interpreter. However, any of the applications we encountered could be rewritten without using an eval-like feature, but would generally be less flexible for adaptation and extension.

4.5 Summary

In this chapter, we have given a general introduction into scripting and scripting languages. We have summarized definitions of other researchers and clarified the terms scripting and scripting language in the context of component-based software development. From our point of view, scripting is a higher-level binding technology for component-based systems whereas a scripting language is a high-level language used to create, customize, and assemble components into a predefined software architecture.

We have analyzed several scripting languages and identified essential and characterizing features. In particular, we have identified two concepts which are essential for scripting: *encapsulation and wiring* and a *foreign code concept*. These two features can be deduced from the main purpose of scripting (i.e. connecting components) and are found in any scripting language. Furthermore, we have discussed a set of characterizing features which position a scripting language in the language design space.

Finally, we have analyzed the essence of "eval" features, a popular mechanism of several scripting languages for executing dynamically created code, and came to the conclusion that any eval-like feature should be considered as a *string-based interface* to a reusable and adaptable *interpreter component*.

Chapter 5

Glue

Closely related to software architectures and scripting is the notion of *glue*: glue is concerned with "putting things together", but the emphasis is on *bridging gaps between incompatible component frameworks*. In an ideal world, there are components available for any task applications have to perform, and these components can be simply plugged together. Since this is not always possible in practice, glue code overcomes *compositional mismatches* and makes components which otherwise cannot be plugged together composable. An example of glue code is a wrapper around a legacy application in order to use this application as a CORBA component. Glue code is often written in languages like Smalltalk (which is good for wrapping legacy code) or C (which is good for gaining access to low-level interfaces). Although scripting languages are marginally concerned with glue in the sense that they can be used to glue together components that have not been designed to work together, typically the hard problems are solved in the stage in which the interface (and interoperability) between the components and the scripting language is defined.

In this chapter, we give a brief overview about glue and glue problems and introduce terms which are important throughout the rest of this work. Second, we analyze glue problems in further detail and define a catalogue of known glue problems. We summarize existing glue technology, and conclude with a discussion of miscellaneous topics related to glue, including requirements for a general-purpose glue language.

5.1 What is glue?

Before introducing important terms related to glue and glue abstractions, reconsider the Python implementation of the keyword extraction script illustrated in section 4.3. Of particular interest in the context of glue are the following lines of code where a set containing all the keywords of the "master" file is defined:

```
KeyWords = Set (FileStream (open (KeyWordFile, 'r'))
               | Filter (lambda line:  (re1.match (line) > 0))
               | Transform (getFirstWord)
               )
```

Analyzing this code, it is immediately possible to identify components and connectors as well as the underlying architecture. The code consists of a data source (i.e. the master file of keywords), four components (i.e. three components of the stream framework `FileStream`, `Filter`, and `Transform` and an instance of the `Set` abstraction), three connectors (two pipes and a method call), and conforms to a *pipe and filter* architectural style [AAG93].

A further analysis of the code reveals other interesting properties:

- The `FileStream` component acts as an *adaptor* for the data source in order to offer the same interface as a stream (of text lines).

- The `Filter` component extracts all the lines of its input-stream which contain a keyword.

- The filter `Transform` acts as a *transformer* since the `Filter` component offers a stream of *text lines*, but the set should only contain the keywords. Therefore, a filter `Transform` is used to extract the first word of each text line (i.e. it transforms the stream of lines into a stream of words).

Therefore, both `FileStream` and `Transform` can be seen as *glue abstractions* since they modify a given component (i.e. its interface and/or its interaction protocol) such that it fits the required behaviour.

Another glue abstraction used in the example is a meta-level feature of Python and is not explicit in the code. The constructor of the `Set` abstraction requires a (possibly empty) *sequence* (e.g. array, list) as a parameter, and it seems that a stream does not fulfill this requirement. However, by appropriately overriding the `__getitem__` method, the class `Stream` is implemented in a way that it can be used in any context where a sequence is required[1] and, therefore, it is possible to directly instantiate a set from a stream. As we will further discuss in section 5.3.5, meta-level abstractions are a powerful mechanism to compose components which are not plug-compatible in the first place.

Summarizing the observations, we can see that *glue techniques are required to adapt components that do not fit the compositional requirements of a framework or system.* Considering this observation, the question arises in what kind of situations a component does not match the compositional requirements of a system. In order to answer this question, reconsider the definition of the term component given in section 2.2 (i.e. a component is a black box entity with a set of *required* and *provided* services). In order to build an application, it is obvious that required and provided services of components have to be plugged together:

[1] As a technical detail: whenever a slice of a sequence is accessed, the `__getitem__` method of the corresponding sequence object is called with successively higher offsets. An `IndexError` is raised if an index is accessed which is outside the range of valid indices. This meta-level feature of Python is used in the `Stream` abstraction: regardless of the actual parameter, `__getitem__` always returns the next item of the stream and raises an `IndexError` when the stream is empty.

> *Software composition* is the process of constructing applications by interconnecting software components through their plugs [ND95].

However, it is not always possible to interconnect components in a desired way: the plugs of two (or more) components may not be *plug-compatible*. Consider the known problem of travellers which are unable to plug the razor they use at home into the plugs of various other countries. In such a situation, *adaptors* are needed to bridge the different interfaces. These kind of problems are often referred to as *architectural mismatch* [GAO95].

Successful composition of components does not necessarily imply successful interoperation:

> *Interoperability* is the ability of software components to communicate and cooperate with each other [Kon95].

Reconsider the problem with the razor: in some countries, different voltages are used and prohibit compatibility even with an adaptor: composition is possible, but interoperability is not. These situations require a *transformer* to transform the incompatible voltage. We will denote these kind of problems as interaction or *interoperability mismatch*.

Both architectural and interoperability mismatch are part of a problem domain which can be denoted as *compositional mismatch* [Sam97]. A compositional mismatch occurs whenever it is impossible to successfully interconnect components with existing connectors.[2]

Based on the discussion and the observations of the example, we define the term glue as follows (a similar definition has been elaborated by D'Souza and Wills in the context of Catalysis [DW99]):

> *Glue* is the part of an application which overcomes compositional mismatches.

The definition of the term glue given above does not necessarily correspond to definitions used in other references. In particular, the notion of glue often refers to any kind of abstraction that can be used to plug components together [All97, Mey98, Lum99]. From our point of view, it is important to make a clear distinction between the notions of *scripting* and *glue*: the former denotes abstractions for *connecting* components whereas the latter makes mismatched components composable.

Closely related to glue are the terms *glue abstraction* and *glue component* [DW99]. As mentioned previously, the emphasis of glue is to bridge gaps between incompatible components. Therefore, any *abstraction* which is used to overcome compositional mismatches should be considered as a glue abstraction, even if it is a component of an existing framework.[3] For the rest of this work, we will use the terms glue, glue abstraction, or glue component interchangeably.

[2] As we will discuss in the next section, architectural and interoperability mismatch are not the only situations where a compositional mismatch may occur.

[3] `FileStream` does not only act as a glue abstraction in the example discussed in this section, but can be seen as a *glue component* since it is part of the stream framework for Python.

5.2 Catalogue of glue problems

In this section, we discuss glue problems at different levels of abstraction and give examples for selected problems. However, we will not discuss other reuse related problems nor give solutions to the glue problems mentioned. A discussion of these topics is part of section 5.3.

Although there is a certain body of knowledge and "best practice" concerning glue and glue abstractions, to our knowledge there exists neither a survey nor a taxonomy which incorporates a wide range of glue techniques. Most surveys concentrate on *component adaptation techniques* such as white-box and black-box approaches [Bos97] or focus on *solutions* of specific problems, but not on the problems themselves [Sha95]. As a first approach for a classification, we therefore discuss glue *problems* at the following levels of abstraction: *platform level*, *interaction level*, *cross-platform level*, and *architectural level*.

Platform level. In section 2.2, we introduced the term *component platform* which indicates any soft- or hardware a component depends on. Several compositional mismatches are due to mismatches in the provided platform a component should be used in: the environment does not offer the right platform a component depends upon. In the following, we list common situations where such a platform mismatch occurs.

- A component is used on the wrong hardware platform (e.g. it is generally not directly possible to execute a DOS executable on a MacOS platform).
- A component is used on the wrong operating system (e.g. components written for DEC VMS cannot be used on OSF/1) or has been developed for a different version of the "same" operating system (e.g. components which heavily depend on kernel routines of UNIX system V cannot be used on a BSD UNIX).
- A component depends on a particular configuration of the underlying storage medium (e.g. a specific directory structure on a hard-disk) and the environment does not reflect the required configuration.
- As discussed in the previous chapter, some scripting languages are dynamically compiled and store the resulting byte-code in separate files (e.g. for any Python module `Module.py` the corresponding byte-code is stored in `Module.pyc`). The byte-code depends on the version of the run-time environment and may not be usable in a different version. As a consequence, a platform mismatch may occur if a component is used in the wrong version of the required run-time environment.
- A component is used on a different windowing system than the one it has been designed for.
- Components distributed in source-code may depend on a specific compiler and cannot be compiled with another compiler.

- Particularly on operating systems that support dynamic linking of libraries, components may depend on a particular version of a library and cannot be used with another version. This kind of platform mismatch can be quite a challenge to solve if the components of an application depend on different versions of the same library.

- In the context of middleware, a component may depend on a particular version of a CORBA ORB and cannot be used on a different ORB. Although the CORBA standard defines interoperability between ORBs (a CORBA component running on one ORB should be able to interoperate with a component running on a different ORB), there are several examples where the inter-ORB interoperability fails [Hel99].

Although this list is certainly not complete, it gives a good insight into compositional mismatches which are due to wrong environments a component should be used in. We argue that a greater part of these problems can be solved by providing the right platform and are not of major interest in the context of a general-purpose composition language.

Interaction level. A second category of compositional mismatches are due to incompatible *interaction protocols* between components. To some extent, these mismatches are due to incompatibilities in the provided and required interfaces of components, and are also referred to as *type matching* problems [Kon93]. However, as discussed below, there are interaction level mismatches which go beyond mismatched interfaces.

- At the lowest level, interface mismatches are due to different *representation* of data: a component interprets data in a different format than the one expected by the corresponding producing component. This can either be at a binary level (e.g. different encodings of integers: little endian vs. big endian), at a representation level (e.g. different layouts for record-like structures), or at an access level (e.g. different range of indices in arrays). Refer to [WWRT91] for a further analysis of representation mismatches.

- Other mismatches at interaction level are due to *syntactical* differences between provided and required services: a provided service is available with a different name than required, the types of the offered parameters are different, the order of parameters differs, or the provided interface requires a different set of parameters.

- There are two types of *semantic* differences: i) due to different programming languages (which we will discuss further below) and ii) due to different programming semantics used in operation call by the caller and callee. As an example of the second type, consider a server component which creates a new object in order to return the result of a request whereas the client expects the result to be returned in specified, existing object (passed as a parameter to the server).

- At a higher level are differences in the way *functionality* is offered and requested. As an example, consider a server that requires multiple calls to invoke a certain service whereas a client makes a single call. This kind of interaction mismatch is also referred to as *protocol mismatch*.

- A last category of interaction level mismatches occur in *concurrent* and/or distributed systems and are generally not due to mismatched interfaces: they are due to i) different synchronization schemes (e.g. synchronous vs. asynchronous service invocation), ii) different assumptions about scheduling of threads, iii) different access patterns to shared resources, iv) different event models, and many more. As an example, consider two components which share a same (external) resource. A service of the "server" component requires exclusive access to shared resource, but this resource is already locked by a client component, which results in a run-time deadlock. Another example of a mismatch of this category are *ownership* problems: a component requires exclusive access to a certain resource (i.e. it "owns" this resource), but another component also accesses this resource at the same time. For further information about interaction level mismatches due to concurrent components, refer to [Lea96].

As mentioned in [Kon93], is is not possible to assign mismatched interaction between components to a single category alone: generally they fall into more than one category.

Cross-platform level. Another category of compositional mismatches is due to components running on different component platforms (e.g. components written in different programming languages, running on different nodes of a distributed system). In the latter case, the mismatches are generally due to different data representations and/or different synchronization schemes. Of more interest are compositional mismatches due to components written in different programming languages.

If two (or more) components written in different languages have to be composed, it is generally not possible to directly access a "foreign" component, but only through a glue layer which acts as a mediator between the languages. Such a glue layer will be denoted as *interlanguage glue* throughout the rest of this work.

The main purpose of interlanguage glue is to act both as an adaptor and transformer for service invocations since the languages involved generally support different data types (names and/or binary representation), different parameter passing conventions (e.g. call-by-value vs. call-by-reference), different behaviour for returning results, and have different exception mechanisms. In addition, there are numerous situations where one of the languages does not offer an equivalent to a (possibly higher-level) abstraction found in another language (e.g. method overloading based on the static types of parameters), and additional glue code is needed in order to map such an abstraction to abstractions found in the other language (if this is possible at all). As a summary, interlanguage glue has to map any kind of abstraction found in one language into abstractions of other languages and, therefore, has to overcome *paradigm clashes* which occur between the different languages. Examples of interlanguage glue are the Java Native Interface [Sun97a] (which maps between Java and C) and LifeConnect [Fla97] (which maps between Java and JavaScript).

However, defining mappings from one language to another is generally not enough in order to safely interoperate between different languages. As an example, consider the run-time system of the ISE Eiffel environment [Mey92] which offers a C API called cecil for accessing and manipulating Eiffel objects from "outside" the run-time system. The run-time system assumes that there is only a single thread running at a time and does not offer any abstractions for synchronizing concurrent accesses. An experiment integrating Eiffel objects into a concurrent Java program (using both JNI and cecil) has revealed that this assumption is essential, and that concurrent accesses to the run-time system result in an unpredictable behaviour, which may even lead to a crash of the system.

The run-time system of the ISE Eiffel environment has an integrated garbage collector which reclaims the memory of all objects that cannot be directly or indirectly accessed by the so-called *root* object of an Eiffel program.[4] This is of particular interest if an object is created from "outside" (using cecil) and not by a "regular" Eiffel object. In order to avoid that these kinds of objects are mistakenly garbage collected, it is possible to *pin* such objects (i.e. pinned objects are not garbage collected). This implies that the interlanguage glue has to ensure correct pinning (and unpinning) of these kind of objects.

Therefore, interlanguage glue should not only define mappings for data-types and abstractions, but has to take care of different behaviour and requirements of the corresponding run-time systems (e.g. synchronization, memory management).

Architectural level. Garlan, Allen, and Ockerbloom analyzed the problems concerning compositional mismatches from an architectural point of view and defined the term *architectural mismatch* [GAO95]. Architectural mismatch stems from mismatched assumptions a component makes about the structure of the system it is to be part of. These assumptions conflict with the assumptions of other components, and are often implicit. Garlan, Allen, and Ockerbloom identify four main categories of assumptions that contribute to architectural mismatch:

- *Nature of components*: this category includes assumptions about the substrate on which a component is built of (i.e. its infrastructure), about which components control the order of computations (control model), and about the way the environment manipulates the data managed by a component (data model).
- *Nature of connectors*: this category contains assumptions about the interaction patterns characterized by a connector (protocols) and about the data being communicated (data model).
- The *global architectural structure* includes assumptions about the communication topology of a system, in particular about the presence or absence of particular components and/or connectors.
- Finally, the *construction process* includes assumptions about the building process of a system.

From our point of view, the definition of the term architectural mismatch is to general and overstresses the notion of an architecture. If two components cannot be directly connected due to a different interface of either the provided or required services (e.g. different

[4]The root object is the first instance of the *root-class* which has to be specified for any Eiffel program.

order of parameters), it is not immediately obvious to call this problem an architectural mismatch; using the term interface mismatch is probably more accurate. The term architectural mismatch should only be used for mismatches caused by conflicting assumptions components make about the *overall architectural structure* of a system. As an example of an architectural mismatch, consider a component of a blackboard architecture which should be used in an unidirectional data-flow architecture.

The first two categories of the list mentioned above roughly correspond to the component platform and interaction-level mismatches, respectively, and have been discussed in previous paragraphs. The last category describes problems due to mutually recursive dependencies of components and can be partially explained by the fact that the case study Garlan, Allen, and Ockerbloom refer to is implemented in C++.[5] We claim that any well-developed system for software composition should enable the specification of such dependencies and solve the corresponding problems automatically.

Versioning. A discussion about glue problems would not be complete if compositional mismatches due to different versions of components were not addressed. These kinds of compositional mismatches are a common source of glue problems, in particular in the context of evolving languages, systems, and frameworks, and can be assigned to any of the categories discussed in this section.

It is a common misunderstanding that version mismatches are mainly due to a different interface of a new version of a component and can be solved by modifying the clients of this component (i.e. adapting service invocations according to the new interface). There are situations where the interface of a new version remains as before, but due to a modification in the underlying language or a different implementation of a service, the *semantics* of a component changes (e.g. blocking rather than returning an error value when reading from an empty buffer). Overcoming these situations may require a deeper understanding how the semantics has changed.

5.3 Glue technology

As discussed in the previous section, glue problems (or compositional mismatches) may occur at different levels of abstraction. In this section, we summarize known glue technology and give solutions to selected glue problems. The reader should note that in general there is more than one way of solving a particular glue problem, but a discussion of all alternatives (including their advantages and disadvantages) is beyond the scope of this work.

Over time, research in software engineering and programming languages has developed a number of techniques for overcoming compositional mismatches, in particular for *adapting* components. These adaptation techniques can be categorized either as *black-box* or *white-box*: white-box techniques focus on adapting a mismatched component by

[5]C++ requires user-defined *forward* declarations in order to resolve mutually recursive dependencies between classes whereas compilers for Eiffel resolve recursive dependencies automatically.

either changing or overriding its internal specification (e.g. inheritance in object-oriented languages) whereas black-box techniques only adapt interfaces [Bos97]. Both techniques have their advantages and disadvantages, and it is a widely accepted fact that white-box techniques require more understanding of a component than just its interface specification. Although there are situations where a white-box approach is feasible, we will not further concentrate on these techniques to overcome compositional mismatches. For the rest of this work, we consider a component as an entity which cannot be directly modified[6] and only consider glue abstractions based on black-box techniques.

5.3.1 Ad-hoc techniques

Software engineers have a wide range of *ad-hoc* techniques for dealing with compositional mismatches in the sense that these techniques solve a very specific glue problem and are neither reusable nor can they be easily generalized. They focus on overcoming compositional mismatches based on a rather local view of the problem and do not take higher-level considerations such as architectural consistencies or intermediate forms into account. In particular, they focus on modifying one of the mismatched components only in order to match the required behaviour, although adapting different components would result in much cleaner solution. In addition, such adaptations are often defined in a context local to the mismatched components and cannot be reused outside this context.

As an example of these techniques, consider a service which is used in two different contexts: one context assumes the name `foo` whereas the other context requires the service under the name `bar`. An ad-hoc solution of the problem is to modify all invocations which require `bar` in order that they refer to the "original" service name.[7] This can be seen as a kind of adapter for "outgoing" services. A cleaner solution of this problem would be to define a (possibly configurable) *adaptor* for `foo` which provides the service also under the name `bar` (i.e. the same service can be invoked using either of the two names). Such a solution is much more reusable and does not require any changes on the client side.

5.3.2 Wrapping techniques

A common technique to overcome compositional mismatches is based on *wrappers* that pack the original component into a new one with a suitable interface. These wrappers usually have the form of an *adaptor*, but there are other glue abstractions which belong to the same category: *bridges*, *proxies*, and *mediators*.

Wrappers. A wrapper implies a form of encapsulation whereby some component is encased with an alternative abstraction and that clients of the wrapped component access the services provided by the wrapper. Wrapping a component can be thought of as yielding

[6] As we discuss in section 5.3.5, there are meta-level approaches for adapting components, which we will denote as *indirect* modification.

[7] Such an adaptation is often done in the source code by modifying the name of a service invocation.

an alternative interface to the component. Wrappers can be further classified into *adaptors* and *transformers*: an adapter bridges incompatible interfaces whereas a transformer is used to modify mismatched interaction protocols. Both wrapper categories have been briefly discussed in section 5.1.

Bridges. A bridge translates between some required assumptions of an arbitrary component to some provided assumptions of another component. The major difference between a wrapper and a bridge is that the repair or glue code of the latter is independent of any particular component. In addition, a bridge must be explicitly invoked by some external "agent", which must not necessarily be one of the components the bridge spans. The last point is intended to convey the idea that bridges are usually transient processes. Bridges typically focus on a narrower range of interface or interaction protocol translations than wrappers do.

Proxies: A proxy provides a surrogate (or placeholder) for another component in order to control access to it and is applicable whenever a there is a need for a more sophisticated reference to a component than a simple wrapper. This is of special interest if the "wrapped" component is located in a different address space: the proxy is responsible for encoding a service request and the corresponding arguments, and hides intercomponent communication protocols. Proxies can be seen as a wrapper which acts both as an adapter and a bridge.

Gamma et al. make the difference between *remote proxies* (which provide a local representative for a component in a different address space), *virtual proxies* (which act as a cache for requests and provide a call-by-need access to a component), and *protection proxies* (which control access to the original component) [GHJV95].

Mediators. Mediators exhibit properties of both wrappers and bridges. The major distinction between a bridge and a mediator is that a mediator incorporates a *planning* function that results in a run-time determination of the corresponding (interface or interaction protocol) translation. Mediators are similar to wrappers as a mediator becomes a more explicit component in the overall software architecture than a bridge, since a bridge can be thought of as incidental repair mechanism whose role in a design often remains implicit [BCK98].

Mediators are a less well-explored technique than either bridges or wrappers. In order to illustrate the concept, consider the scenario of assembling a sequence of bridges in order to integrate components whose specific integration requirements arise at run-time. One of the components produces data in a format D_0, another requires data in format D_2, and there is no direct bridge from D_0 to D_2. However, there are separate bridges which translate from D_0 to D_1 and from D_1 to D_2, respectively. The mediator in this scenario assembles the alternative bridges in order to complete the translation from D_0 to D_2. The reader should note that UNIX filters are often used in this type of scenario.

In contrast to the ad-hoc mechanisms discussed previously, wrappers, bridges, proxies, and mediators are higher-level abstractions for overcoming compositional mismatches and can in general be reused and generalized more easily. They still focus on a rather local view of problems and neglect higher-level considerations. One should note that wrapping techniques cannot always be applied to overcome compositional mismatches [YS97].

5.3.3 Intermediate forms

In contrast to the glue techniques discussed previously, *intermediate forms* focus on a different approach: adapting *all* components of a system in a way that they conform to some standard form. Such a standard (or intermediate) form is generally based on one of the foreign code concepts discussed in section 4.2. Whereas wrappers and other glue techniques mainly focus on *overcoming* compositional mismatches, standard forms try *avoid* them (at least to a certain degree) by restricting the kind of components which can be used in a system. Standard forms generally specify i) how interfaces for components have to be defined, ii) what kind of data entities can be exchanged between components, iii) what kind of interaction mechanisms and iv) what kind of architectural style(s) can be used. Note that applications based on intermediate forms tend to focus on specific application domains or architectures.

A popular example of an intermediate form are software buses. A *software* bus (comparable to a hardware bus used in most modern computer systems) defines a standardized communication protocol for exchanging data (i.e. a set of data types that can be used to exchange data and a number of service invocation mechanisms), takes care of a correct message handling, and performs necessary data conversions. From a different perspective, a software bus can be seen as a kind of intelligent blackboard. Examples of such software buses are Bart [Bea92] and POLYLITH [Pur94].

The concept of an *Object Request Broker* (which is often referred to as *middleware*) goes a step further than software buses. Middleware does not only define interface restrictions for components and interaction protocols, but offers additional services such as event models, transactions, and service traders. Examples of middleware are both CORBA [OMG96] and COM [Rog97].

Other examples of intermediate forms are special file formats such as RTF, MIF, or Postscript, network services such as HTTP or FTP, or the Java Virtual Machine [LY96].

5.3.4 Patterns

As discussed in section 3.2, patterns describe the solution of a recurring (object-oriented) design problem in relationship to its context. Due to the fact that most of these patterns focus on *object composition*, and not on inheritance, they are not restricted to object-oriented programming languages and can be adapted for component-oriented software development in general.

Although patterns focus on solving *design problems*, we argue that some of them can be applied in a context where a compositional mismatch occurs. As an example, consider the Adaptor pattern which can be applied in situations where a provided interface of a class does not match the required interface of its clients: "Adapter lets classes work together that could not otherwise because of incompatible interfaces" [GHJV95].

There are several other examples of patterns which focus on briding incompatible class hierarchies, adapting incompatible interfaces, or extending behavioural properties. Due to the fact that they describe general glue technology in a context/solution form and provide a common vocabulary for understanding glue problems, we denote such patterns as *glue patterns* for the rest of this work.

Examples of such glue patterns are the Adaptor, Bridge, Mediator, and Proxy patterns which describe variations of the concepts discussed in section 5.3.2 [GHJV95]. Other patterns focusing on similar concepts are the Facade, Strategy, and Decorator patterns or the Object Wrapper pattern [MM97]. Patterns describing architectures based on intermediate forms are the Microkernel, Broker, and Blackboard patterns discussed in [BMR+96].

5.3.5 Reflective approaches

As mentioned before, many glue techniques are based on *wrappers* that pack the original component into a new one with a suitable interface. If wrappers are used frequently, this technique gives rise to serious performance problems [Höl93]. In this section, we discuss different approaches based on *meta-level abstractions* [McA95] which try to overcome some of the problems of wrapper technology.

One of these approaches is based on the idea of *intercepting service invocations* and to manipulate them according to the requirements of the receiver of the corresponding invocation. In particular, if service invocation is based on *message sending* (as it is the case in object-oriented programming languages), messages can be transformed, delayed, or even delegated. Comparable approaches have been used in existing languages (e.g. Sina [Aks89], CLOS [ABB+89], and Smalltalk [GR89]), but none of these approaches primarily focuses on overcoming compositional mismatches.

As an example of connecting components using intercepted service invocations, consider the concept of *executable connectors* [Duc97]. Connectors are run-time entities which define a set of rules how connected objects react and interact during external message sending.[8] They change the observable behaviour, without modifying the objects themselves. Therefore, connectors act as a higher-level glue abstraction for composing and synchronizing objects.

Another meta-level approach is based on the concept of explicit contexts or *namespaces*, which has been introduced in section 4.4. A namespace provides its own mechanism for *name lookup* and defines a (partial) mapping from names to values. Some programming languages offer meta-level abstractions for controlling, modifying, or extending the name lookup mechanism of namespaces (e.g. the method `__getattr__` in Python used to perform attribute lookups can be overridden). Namespaces can be used to transparently access remote components, as *gateways* to other languages, and as a synchronization mechanism. Furthermore, namespaces can be composed (or nested) in order

[8] The term *connector* is used differently here than in the discussion about software architectures in chapter 3.

to allow for a flexible access to hierarchical name structures. The usage of namespaces with explicit control offers other benefits in the context of composition as well, but a detailed discussion is beyond the scope of this work.

5.4 Discussion

In the following, we will briefly discuss selected issues related to glue and glue abstractions. In particular, we focus on limitations of wrapping techniques and requirements for reusable and adaptable glue abstractions. It is obvious, however, that there are other aspects related to glue and glue abstractions which are also relevant (e.g. run-time performance, network protocols), but a detailed discussion of all these aspects is beyond the scope of this work. For further information, refer to [MMM95] or [YS97].

Limitations of wrapping techniques: In [Höl93], Hölzle discusses problems related to wrapping techniques in the context of integrating components in object-oriented programming languages. From his point of view, wrapping techniques give rise to serious performance problems when they are heavily used in applications, since all interactions with wrapped components have to go through the wrapper. Additional redundancy is introduced into systems since wrappers generally duplicate part of the interfaces of components, which leads to additional changes when the interface of a wrapped component is modified, unless clever compile- or run-time support is provided. Applications using wrappers are often harder to understand and, therefore, harder to extend and maintain. As a consequence, wrapping techniques should be used with care and can often be replaced by meta-level abstractions.

Yellin and Strom formalize the notion of an adapter as a bridge between functionally compatible components with type- and protocol-incompatible interfaces [YS97]. From their point of view, an adaptor is a *finite-state machine* that has interfaces to the components that need to collaborate. The behaviour of the adapter is defined by its transition rules and contains a *finite* set of memory cells to buffer data for multiple service invocations.[9] Based on these assumptions, there are protocol mismatches which cannot be handled since they would require an infinite set of memory cells. For further information about these protocol mismatches, refer to the original reference.

Requirements for glue abstractions: In an ideal world, there are components available for any task an application has to perform and these components can simply be plugged together. However, this ideal scenario has not yet become true, and there are many situations where a component with almost the required functionality is available, but it cannot be integrated into a system due to mismatched assumptions. Therefore, a general-purpose composition language must offer support for overcoming these compositional mismatches: it must offer *reusable glue abstractions*.

[9] As an example, consider the situation where a client invokes a service with a single invocation whereas the server requires multiple calls.

In the previous sections, we identified compositional mismatches at several levels of abstraction and discussed techniques to overcome these mismatches. In the following, we discuss a set of requirements for reusable glue abstractions a general-purpose composition language should fulfill. These requirements also provide a framework for comparing and selecting glue abstractions.

In [Bos97], Bosch defines a set of requirements in the context of *component adaptation techniques*, which do not only help to classify, but also give insight how new adaptation techniques should be designed. Although the requirements primarily focus on adaptation, they can be applied to any kind of glue abstraction. Therefore, we discuss these requirements from a slightly different point of view than in the original reference.

- Glue abstractions should be as *transparent* as possible in the sense that none of the involved components is aware of the inserted glue code. In addition, all services that match the required behaviour and/or protocol should not be affected by glue.
- In order to ensure a transparent exchange of components, a glue abstraction should only know as much of the internal structure of the involved components as is needed for overcoming the corresponding compositional mismatch. Ideally, glue abstractions should only depend on interfaces of components, which implies that glue abstractions should be based on *black-box* techniques.
- Any glue abstraction should be defined in a way that it can be applied to components without redefining them (i.e. easy composition of glue and components). In addition, glue abstractions should be *composable* themselves (i.e. it is possible to compose glue abstractions) and a composition of glue abstractions should again be a glue abstraction.
- Glue abstractions generally consist of a *generic* and a *specific* part. The specific part is fixed and cannot be changed, but the generic part should be sufficiently *configurable* in order to enhance the reusability of a glue abstraction. A general problem with many glue abstractions is the fact that the generic part cannot be separated from the specific part, which results in a smaller degree of *flexibility* and *reusability*. Hence, a glue abstraction should improve reusability by making a clear separation of the two concerns.

This list covers the important requirements for reusable glue abstractions. However, as we pointed out in section 2.4, glue abstractions tend to be ad-hoc and lack a well-defined formal semantics in order to reason about compositions of components. Therefore, it is necessary to add an additional requirement for glue abstraction: their semantics should be defined in terms of well-defined *formal basis*.

Chapter 6

Unification of concepts

As discussed in chapter 2, object-oriented programming alone is not enough to guarantee the development of flexible systems, but it provides a good set of tools and techniques that can be used for component-based application development. Components, however, are not enough either, since a component without an architecture is like a Lego – all by itself. CORBA, Delphi, JavaBeans, and D-Active-COM-X-++ are also not enough – each solves important technical problems, but does not go beyond a specific domain.

In the previous chapters, we have discussed several important concepts and techniques related to software composition and proposed a conceptual framework for software composition in which five of these techniques are combined, namely:

- *component frameworks* provide software components that encapsulate useful functionality,
- *architectural description languages* explicitly specify architectural styles in terms of interfaces, contracts, and composition rules that components must adhere to in order to be composable,
- *scripting languages* are used to specify compactly and declaratively how software components are plugged together to achieve some desired result,
- *glue abstractions* adapt components that need to bridge compositional mismatches,
- *coordination models* provide the coordination media and abstractions that allow distributed components to cooperate.

In this chapter, we illustrate how the concepts of components, scripts, and glue are used in practice, followed by a discussion about the conceptual framework based on different points of view.

6.1 Concepts in practice

In order to illustrate the concepts of components, architectures, scripts, and glue, consider the following example (originally presented in [SN98]): the University of Berne provides

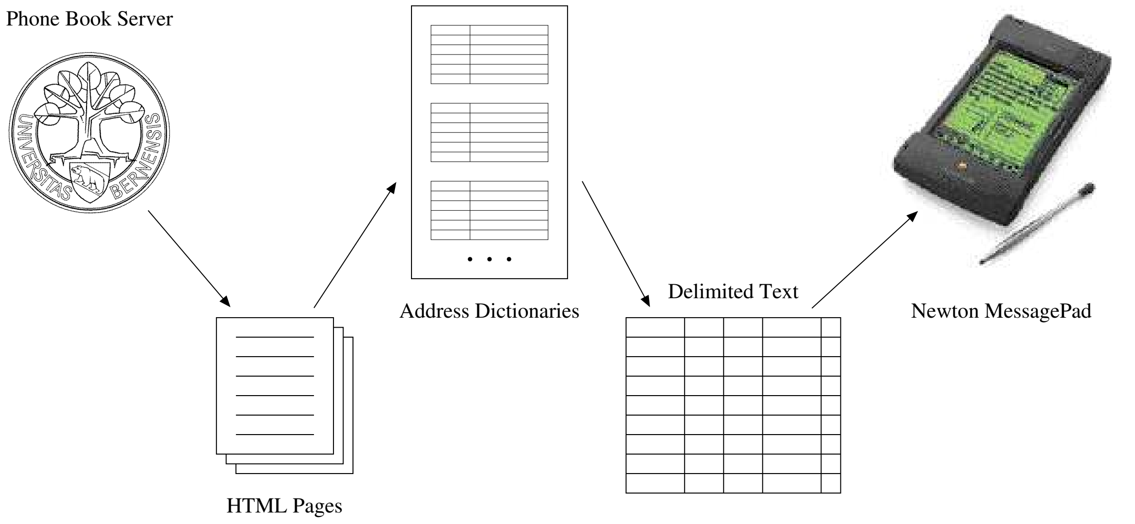


Figure 6.1: Schema of UniBe phone book application.

an online phone book with information about people working at the university. The online phone book has a web-interface (i.e. it can be used by any kind of web-browser) which is not ideal for interchanging information with other applications, for example the Newton MessagePad Names application. In order to interchange information with other applications, we illustrate the design and implementation a new application which

1. offers an interface for user input (i.e. search strings),
2. connects to the phone book server and downloads the query results of the corresponding queries, and
3. parses the resulting HTML page and generates a table of delimited text (or any other desired format) which can be imported by the Names application.

In order to enhance reusability and extensibility of this application, the parsing step should be further subdivided into i) a parser which parses the HTML pages and generates a dictionary for each address found and ii) a formatter which formats the dictionaries into the desired format.

Analyzing the requirements, it becomes clear that a *data-flow architecture* is a natural approach for implementing this application. The query results can be seen as data sources, which are parsed, filtered (the HTML pages contain other information not directly related to addresses of people), transformed into address dictionaries and, finally, formatted into the desired format (such as delimited text). Therefore, the application may consist of stream, filter, and transformer components. For a schema of the application, refer to Figure 6.1.

Due to the fact that we have previously implemented a framework for stream processing in Python (see section 4.3 for further details), it is a natural choice to implement the application using this framework. In addition, there exists several Python packages providing various World-Wide-Web services, in particular for opening and reading URL's [WvRA96]. Therefore, the major part of the application can be implemented by appropriately configuring the components of the corresponding frameworks. The only application-specific abstractions which have to be implemented are i) address dictionaries `Address` (which know how to format themselves in different formats) and ii) an HTML parser `ParseAddr` which transforms a stream of lines (i.e. the output of the queries) into a stream of `Address` objects. In order to use the HTML parser as a component of the stream framework (i.e. it has to read from and write to a stream), it must offer the same interface as the other filter or transformer components.[1] The parser itself uses components of the regular expression package in order to parse and format its input, and the address dictionaries are implemented as an extension of the class `UserDict` (which is a wrapper class around the built-in Python dictionaries).

The problem with this approach is that components from different frameworks have to be connected which are not plug-compatible: the result of the address queries (a list of HTML pages) does not correspond to a single character stream which the parser uses as its input. In order to solve this problem, it is necessary to write glue code which transforms the HTML pages into a single character stream. The stream framework already offers some abstractions for this transformation: i) `FileStream` transforms a file-like object (i.e. an object which offers the same interface as a `file` object) into a stream and ii) `CatStream` concatenates a list of streams into a single stream. The only application-specific glue code which has to be written is an abstraction (i.e. the function `ubtb`) which transforms the result of a single query into a file-like object.

For the source of `ubtb` and the top-level function `main`, refer to Figure 6.2. All the other abstractions used in the application (including the stream framework) are given in appendix A.

An analysis of the design and implementation process for this application reveals that it conforms to the conceptual framework we proposed in this part and that it is possible to identify components, architectures, a script, and glue abstractions. In particular, the analysis reveals the following:

- The application conforms to a pipe and filter architectural style, and the architecture is made explicit in the source code (i.e. in the top-level script).
- The pipe operator '|' is used as a high-level connector (it connects streams with filters and/or transformers) and adopts an algebraic view of composition.
- The application reuses existing components and connectors: (text-)streams, string dictionaries, url/http components (i.e. components of the `urllib` framework), and regular expression components to parse text.

[1]Due to the fact that the stream framework is object-oriented, `ParseAddr` inherits from an abstract class `InputStream` which defines the interface and default behaviour for pipe-and-filter composition of streams.

```
def ubtb(name):
    # Return a file-like object containing the HTML of the query
    # result for "name":
    from urllib import urlopen              # import www services
    try:
        name = string.join (string.split (name, ' '), '+')
        url = urlopen("%s?name=%s" % (ubtbURL, name))
    except:
        sys.stderr.write ("Can't open " + ubtbURL)
        sys.exit(1)
    return url

def main():
    # default format -- use the Address.display() method:
    format = Transform (lambda a:  a.display())
    # Open a ubtb url "file" for each arg, convert it to a FileStream,
    # concatenate streams, parse them into addresses, and format them:
    source = CatStream (map (FileStream, map (ubtb, args)))
    parser = ParseAddr()

    print source | parser | format          # explicit architecure
```

Figure 6.2: Python source code of UniBe phone book application.

- The application consists of components of several frameworks and, therefore, glue abstractions are needed in order to overcome compositional mismatches: `ubtb` wraps a HTML query and returns a file-like object, `FileStream` transforms file-like objects into text streams, `CatStream` concatenates a list of streams into a single stream etc.

- User-defined (application-specific) components are compositions of (lower-level) components (e.g. streams of `Address` objects).

- The application uses abstractions of both object-oriented (e.g. classes and objects) and functional programming (e.g. lambda expressions).

The reader should note that the HTML parser is the most “fragile” component of the application since any change in the output format of the query results may require a corresponding adaptation in the parser. However, none of the other components nor the overall architecture of the application are affected by a change of the output format and, therefore, do not have to be adapted. If a user requires the query results in a different format than the one provided by the application, it is easily possible to exchange the `format` component by a new one, and none of the other components are affected by this modification, either.

This fact also holds for testing: in order to test the parser component, it was possible to replace the stream `source` of the function `main` with a stream-like object which does not connect to the web-server, but directly returns the contents of a test-file (containing the results of test query). Again, none of the other components were affected.

Summarizing the observations made during the development of this application, we can say that application development using the conceptual framework proposed in this part leads to an adaptable and extensible application with a well-defined, robust architecture.

6.2 Discussion

The concepts we have discussed in the previous chapters define a framework for *composing* applications from component frameworks and reflect the main requirements with have discussed in section 2.5: *making a clear separation between computational and compositional elements*.

As mentioned in the previous section, using the conceptual framework leads to adaptable and extensible applications with a well-defined architecture. In particular, applications can evolve by i) adapting existing components (reconfiguration of required services), ii) extending existing components, iii) adding new components which are compatible with the "old" ones, iv) integrating external components (using glue abstractions), and v) reconfiguring the connections between components.

One might argue that these concepts only apply to run-time composition, as scripting languages are typically dynamically compiled or interpreted. The same ideas, however, apply equally well to compile-time composition. Consider, for example, the Standard Template Library (STL) [MS96]. STL provides a set of C++ container classes (such as vectors, lists, sets etc.) and template algorithms for common kinds of data manipulations on the container classes (e.g. searching, sorting, merging). STL has all the properties we have previously established for component frameworks: it focuses on component composition rather than white-box reuse, it incorporates a collection of reusable components, fixes the interfaces components may have, and defines a set of rules how components can be composed. All applications using STL, therefore, share a common architectural style, even though the concrete architecture may not always be explicit.

The keyword extraction script illustrated in section 4.3 may be implemented in C++ using similar concepts to those we have already used in the Bourne Shell scripts: components (STL containers), connectors (generic functions), and glue (e.g. input/output stream adapters to make `cin` and `cout` look like containers). The major difference between the C++ program and the Bourne Shell script is that i) a C++ program does not make the underlying architecture of the program explicit, and ii) any C++ program using STL only works in a sequential environment and, therefore, does not require any coordination abstractions.

Another important aspect related to the conceptual framework we have presented is the fact that it cannot only be used for building applications, but also for *documenting* existing applications. This is of particular interest in the context of reengineering legacy systems. In parallel to our research on a composition language and system, the Software

Composition Group participates in FAMOOS,[2] a European industrial research project on reengineering object-oriented legacy systems towards component-based frameworks. Early adopters of object-oriented technology now find themselves with large, object-oriented applications that are critical to their business interests, but are difficult to adapt to changing business needs. Results show that a *pattern-based approach* is most promising, since similar reengineering problems seem to recur across applications. In particular, these patterns aim at identifying components, architectures, and compositions in legacy applications.

The conceptual framework we have proposed in this work mainly focuses on technical issues. However, if it should be applied in a broader context of component-based software development, there are several *methodological issues* which must be addressed as well. In the following, we briefly outline some of theses issues.

- Given a particular design problem, how do we select a suitable architectural style that leads to a flexible and extensible design?
- How do we drive application development from a given architectural style and/or component framework?
- Given a problem domain and a body of experience from several applications, how do we re-engineer existing software into a component framework?
- During the development of a component framework, how do we select a suitable architectural style to support black-box composition?

Although we do not yet pretend to have answers to all these questions, we think that separating applications into components and scripts is an essential step towards a methodology for component-based software development.

[2]FAMOOS is an industrial ESPRIT Project (No 21975) in the IT Programme of the Fourth ESPRIT Framework Programme.

Part III

Towards a composition language

As discussed in Part I, software composition is supported in a rather ad-hoc way in existing languages and systems. In order to overcome these problems, we identify the need for a rigorous semantic foundation (i.e. a *composition calculus*) for specifying applications as compositions of software components [NSL96]. In particular, if we can understand all aspects of software components and their composition in terms of a *small set of primitives*, then we have a better hope of being able to cleanly integrate all required features in one unifying concept. However, we are not seeking for a composition calculus which incorporates all required abstractions as primitives, but we rather define a minimal calculus where all compositional abstractions can be expressed in terms of the primitives of the calculus.

The requirements for a composition language given in section 2.5 suggest an approach in which we use a formal process calculus as the core of our formal foundation and to define higher-level abstractions in terms of the core calculus. Such an approach has already been used for the PICT programming language where all language features are defined by syntactic transformation to the mini π-calculus [PT97]. Although the mini π-calculus can be used to model concurrent objects and composition mechanisms [LSN96, SL97], it is inconvenient for modelling general composition abstractions due to the dependence on positional parameters in communications. For example, a generic Readers-Writers synchronization policy cannot be directly coded without wrapping method arguments in order to treat an arbitrary number of arguments as a single value [Var96].

Dami has tackled a similar problem in the context of the λ-calculus and has proposed λN, a calculus in which parameters are identified by *names* rather than positions [Dam94, Dam98]. The resulting flexibility and extensibility can also be seen in HTML forms, whose fields are encoded as named (rather than positional) parameters in URLs, in Python, where functions can take arguments by *keywords* [vR96], and in Visual Basic, where *named arguments* can be used to break the order of possibly optional parameters [Mic97]. Based on these ideas, Lumpe has defined the $\pi\mathcal{L}$-calculus, where the communication of names or tuples of names of the π-calculus is replaced by communication of *forms* [Lum99]. Furthermore, the $\pi\mathcal{L}$-calculus introduces the concept of *polymorphic form extension* as a mechanism to compose arbitrary forms.

Although the $\pi\mathcal{L}$-calculus seems to be a suitable formal foundation for a composition language, experiments have shown that there are problems and contexts where the expressive power of the $\pi\mathcal{L}$-calculus is not enough, and we identified the need to extend the calculus according to the corresponding shortcomings. Therefore, we extend the $\pi\mathcal{L}$-calculus with the concepts of *polymorphic restriction* and *matching*, which leads to the definition of the FORM calculus. We analyze the expressive power of the FORM calculus by modelling higher-level abstractions in order to validate the suitability of the FORM calculus as a minimal semantic foundation for a general-purpose composition language.

Part III is organized as follows: in chapter 7, we define the FORM calculus, an extension of the π-calculus, which forms the basis for the rest of this work. We use the FORM calculus to define a meta-level framework for concurrent, object-oriented programming (chapter 8) as well as for modelling compositional abstractions (chapter 9). We conclude with a summary of the main observations in chapter 10.

Chapter 7

Towards a composition calculus

There are several plausible candidates as computational models for objects, components, and software composition in general. The λ-calculus has the advantage of having a well-developed theoretical foundation and being well-suited for modelling encapsulation, composition and type issues, but has the disadvantage of saying nothing about concurrency or communication [CW85]. Process calculi such as CCS have been developed to address just these shortcomings [Mil89]. Early work in modelling concurrent objects has proven CCS to be an expressive modelling tool, except that dynamic creation and communication of new communication channels cannot be directly expressed and that abstractions over the process space cannot be expressed within CCS itself, but only at a higher level [Pap92].

The π-calculus, a calculus in which the topology of communication can evolve dynamically during evaluation, addresses these shortcomings by allowing new names to be introduced and communicated much in the same way that the λ-calculus introduces new bound names [MPW92]. This is needed for modelling creation of new objects with their own unique object identifiers. The basic (monadic) calculus allows only communication of channel names. The polyadic π-calculus supports communication of tuples, needed to model passing of complex messages [Mil91]. The higher-order π-calculus supports the communication of process abstractions, which is needed for modelling software composition within the calculus itself [San93]. Interestingly, the polyadic and higher-order variants of the π-calculus can be faithfully translated (or "compiled") down to the basic calculus, so one may confidently use the features of richer variants of the calculus knowing that their meaning can always be understood in terms of the core calculus. The π-calculus has previously been used by Jones [Jon93], Vasconcelos [Vas94], Barrio [BS95], and Walker [Wal95] to model various aspects of object-oriented programming languages.

A simplification of the π-calculus has been studied by Honda and Tokoro [HT91] and Boudol [Bou92], who proposed that asynchronous communication provides a better foundation for distributed systems, without any loss of expressive power. Sangiorgi extended the proposal by allowing polyadic communication [San95]. This variant (also known as the *mini π-calculus*) essentially forms the core language for PICT [PT97] and is the starting point of the work presented in Part III.

Several other computational models suggest themselves as candidates, but each lacks the simplicity of the π-calculus. The actor model [Hew77, Ahg86] is one of the earliest formalisms used for modelling concurrent objects, but its semantic foundations are still under development [AMST93]. Most aspects of actors can easily be modelled in the π-calculus (except for fairness). Rewriting logics [Mes90] are an attractive basis, but can also be seen as a meta-language for defining calculi like the π-calculus. Linear logic has also been used to model concurrent objects [AP90], but the path to an operational view of objects as communicating processes is not very direct.

In this chapter, we give an informal introduction into the polyadic mini π-calculus, the starting points of our study, followed by a brief discussion about the $\pi\mathcal{L}$-calculus, a variant of the π-calculus, which replaces the tuple-communication with communication of *forms*. The major part of this chapter is dedicated to the FORM calculus, an extension of the $\pi\mathcal{L}$-calculus based on our results of modelling compositional abstractions. We conclude this chapter with a comparison of the $\pi\mathcal{L}$-calculus and the FORM calculus.

7.1 The polyadic mini π-calculus

In this section, we informally introduce the *polyadic mini π-calculus* which is the basis for our basic object encodings introduced in section 8.1 and the starting point of our study. Readers familiar with the mini π-calculus may skip this section. For further information about the mini π-calculus, refer to [San95].

The polyadic mini π-calculus is built from the operators of inaction, input prefix, output, parallel composition, restriction, and replication. Small letters $a, b, ..., x, y, ...$ range over the infinite set of channels or *names*,[1] and $P, Q, R, ...$ over the set of processes:

$$P ::= \mathbf{0} \;\Big|\; a(\tilde{x}).P \;\Big|\; \overline{a}\langle\tilde{x}\rangle \;\Big|\; P_1|P_2 \;\Big|\; (\nu\ a)P \;\Big|\; !a(\tilde{x}).P$$

0 is the inactive process. An input-prefixed process $a(\tilde{x}).P$, where $\tilde{x} = x_1, \ldots, x_n$ has pairwise distinct names, waits for a tuple of names $\tilde{y} = y_1, \ldots, y_n$ to be sent along a and then behaves like $P\{\tilde{y}/\tilde{x}\}$, where $\{\tilde{y}/\tilde{x}\}$ is the simultaneous substitution of names $\tilde{x}$ with names $\tilde{y}$. An output $\overline{a}\langle\tilde{x}\rangle$ emits names $\tilde{x}$ at a. Parallel composition runs two processes in parallel. The restriction $(\nu\ a)P$ makes name a local to P. A replication $!a(\tilde{x}).P$ stands for a countably infinite number of copies of $a(\tilde{x}).P$ in parallel. Finally, input prefix, restriction, and replication have precedence over parallel composition. In order to avoid ambiguous process definitions and enhance readability, parentheses may be used to group processes.

In an input prefix $a(\tilde{x})$ and an output prefix $\overline{a}\langle\tilde{x}\rangle$, we call a the *subject* and $\tilde{x}$ the *object* of a communication. Additionally, we will often use the special name '_' as a wild-card symbol. Values bound to this name are unimportant for the following process and will be ignored. Furthermore, a process sending an empty tuple along a channel x is abbreviated as $\overline{x}$.

[1] Throughout this work, we will use the words *name* and *channel* interchangeably.

$$\text{PAR}: \frac{Q \longrightarrow R}{P \mid Q \longrightarrow P \mid R} \qquad \text{RES}: \frac{P \longrightarrow Q}{(\nu\, x)P \longrightarrow (\nu\, x)Q}$$

$$\text{STRUCT}: \frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q}$$

$$\text{COM}: a(\tilde{x}).P \mid \overline{a}\langle\tilde{y}\rangle \longrightarrow P\{\tilde{y}/\tilde{x}\}$$

Table 7.1: Reduction rules for the polyadic mini π-calculus.

The set of *free names* $\text{fn}(P)$ and the set of *bound names* $\text{bn}(P)$ of a process P are defined in the usual way. The binding operators for names are the input prefix $a(\tilde{x})$ (which binds $\tilde{x}$) and the restriction $(\nu\, x)$.

The semantics of the polyadic mini π-calculus is presented using a *reduction semantics*. This style of semantics involves defining two relations on processes: a *reduction relation*, specifying the actual communication behaviour of processes, and a *structural congruence relation*.

The reduction relations given in Table 7.1 describe the reduction of polyadic mini π-terms. The first two rules state that we can reduce under both parallel composition and restriction, respectively. The symmetric versions of both rules can be omitted due to structural congruence (see below). The communication rule takes two processes which are willing to communicate along a channel a and simultaneously substitutes the free names $\tilde{x}$ with names $\tilde{y}$. This rule enables that restricted names may be communicated form a process P ot another process Q (*scope extrusion* of restricted names). Note that the communication rule is the only rule which directly reduces a π-term and requires that processes are in a particular format. The structural congruence rules allow us to rewrite processes such that they have the correct format required by the communication rule.

The structural congruence relation is the smallest congruence relation over processes that satisfy the axioms below:

$$!a(\tilde{x}).P \equiv a(\tilde{x}).P \mid !a(\tilde{x}).P$$

$$P \mid Q \equiv Q \mid P \qquad P \mid \mathbf{0} \equiv P$$

$$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

$$(\nu\, x)P \mid Q \equiv (\nu\, x)(P \mid Q) \quad x \notin \text{fn}(Q)$$

The replication operator enables processes to have an "infinite behaviour". A replicated process is not "consumed" after matching an output, but persists in parallel with the instantiation of its body. Such a process can be seen as a server, and its structure is of such a general nature that it is helpful to have a higher-level syntax for it. A similar derived form is also a basic element in the language PICT [PT97].

$$\mathbf{def}\ X[\tilde{x}] = P\ \mathbf{in}\ Q \stackrel{def}{=} (\nu\ X)(!X(\tilde{x}).P \mid Q)$$

Milner has demonstrated how data structures could be encoded in the π-calculus [Mil91]. An encoding of booleans can be defined as follows:

$$\begin{array}{rcl} \mathbf{def}\ \mathit{True}[r] & = & (\nu\ b)(\overline{r}(b) \mid !b(t,f).\overline{t}) \\ \mathbf{def}\ \mathit{False}[r] & = & (\nu\ b)(\overline{r}(b) \mid !b(t,f).\overline{f}) \end{array}$$

A boolean value is a channel along we send/receive two channels for the next *true* and *false* interaction. The processes True and False do not take any parameter other than a result channel r. They both create a new channel b that serves as the location of the boolean value and return b along the result channel r. Furthermore, since True and False are replicated processes, they can answer queries about a boolean value b more than once. If we had omitted the replication, the resulting processes would yield *linear* booleans.

Negation, disjunction, and conjunction of the boolean encoding presented above can be defined as follows:

$$\begin{array}{rcl} \mathbf{def}\ \mathit{Not}[b,r] & = & (\nu\ t,f)(b(c).\overline{c}\langle t,f\rangle \mid t(\_).\mathit{False}(r) \mid f(\_).\mathit{True}(r)) \\ \mathbf{def}\ \mathit{And}[b_1,b_2,r] & = & (\nu\ t_1,t_2,f)(b_1(c_1).\overline{c_1}\langle t_1,f\rangle \mid t_1(\_).b_2(c_2).\overline{c_2}\langle t_2,f\rangle \\ & & \quad \mid t_2(\_).\mathit{True}(r) \mid f(\_).\mathit{False}(r)) \\ \mathbf{def}\ \mathit{Or}[b_1,b_2,r] & = & (\nu\ t,f_1,f_2)(b_1(c_1).\overline{c_1}\langle t,f_1\rangle \mid f_1(\_).b_2(c_2).\overline{c_2}\langle t,f_2\rangle \\ & & \quad \mid t(\_).\mathit{True}(r) \mid f_2(\_).\mathit{False}(r)) \end{array}$$

The process Not takes two arguments: i) a channel b that serves as the location of the original boolean value and ii) a result channel r along Not returns the location of the negated boolean value. Internally, Not creates two new channels t and f and sends them along b. In parallel, Not starts two processes that listen at t and f, respectively. If the boolean value signals at t (i.e. the interaction for value true), then Not returns along channel r the value False. If the boolean value signals at f (i.e. the interaction for value false), then Not returns along channel r the value True. The encodings for And and Or are similar.

7.2 The $\pi\mathcal{L}$-calculus

In this section, we briefly introduce the $\pi\mathcal{L}$-calculus, a conservative variant of the π-calculus [Lum99]. The main idea behind the $\pi\mathcal{L}$-calculus is to replace the communication of names or tuples of names by communication of so-called *forms*, a special notion of extensible records. More precisely, in the $\pi\mathcal{L}$-calculus the polyadic tuple communication of the mini π-calculus is replaced by monadic communication of forms. The communication of forms is motivated by the fact that interaction based on names leads to more adaptable and extensible components than interaction based on positions. As a consequence of form communication, unlike in the mini π-calculus, constants and variables in the $\pi\mathcal{L}$-calculus are strictly distinguished. In fact, even though it is possible to communicate names by means of binding from labels to names, names are always immutable.

However, the $\pi\mathcal{L}$-calculus is more than just replacing polyadic tuple communication by monadic communication of forms. It also introduces so-called *polymorphic form extension*, a concept that corresponds to asymmetric record concatenation [CM94], which is one of the key features for modelling higher-level compositional abstractions. In fact, polymorphic form extension is a powerful mechanism for software composition, since it enables the composition of arbitrary services in order to achieve the required behaviour (see also section 8.3 for further details).

The following example illustrates several key concepts of the $\pi\mathcal{L}$-calculus, and shows how it facilitates the definition of extensible higher-level abstractions.[2]

$$
\begin{aligned}
\mathbf{def}\ \mathit{dispatch}(X,Y,r) \quad &= \quad (\nu\ b, r_1, r_2)(\overline{r}(\langle \mathit{close}=b\rangle) \\
&\qquad\qquad \mid\ !b(Z).(\overline{X_{\mathit{close}}}(\langle \mathit{result}=r_1\rangle) \\
&\qquad\qquad\qquad\qquad \mid \overline{Y_{\mathit{close}}}(\langle \mathit{result}=r_2\rangle) \\
&\qquad\qquad \mid r_1(\_).r_2(R).\overline{Z_{\mathit{result}}}(R))) \\
\mathbf{def}\ \mathit{fixcomp}(L,R,r) \quad &= \quad (\nu\ d)(\mathit{dispatch}(L,R,d) \\
&\qquad\qquad \mid d(C).\overline{r}(\langle p=L_p\rangle\langle s=R_s\rangle\langle d=R_d\rangle\langle \mathit{close}=C_{\mathit{close}}\rangle))
\end{aligned}
$$

This example defines an abstraction $\mathit{fixcomp}$ which composes two forms, say a GUIList and a Multiselector. The GUIList component has two services p (for paint) and $close$ whereas the Multiselector offers the services s, d (for select and deselect), and $close$. A composition of these two components offers the union of both sets of services, and, in order to close the composite component correctly, an invocation of $close$ must be delegated to both components (using the $dispatch$ abstraction). Note that $\mathit{fixcomp}$ explicitly refers to the four services of the composite component in order to define the composition.

Although $\mathit{fixcomp}$ defines a correct composition of a GUIList and a Multiselector component, it cannot be used in a context where the composition should also provide extensions of the two components. For example, the GUIList component may be extended with a $resize$ service or the Multiselector component may define a new service $selectAll$. If we compose such components with $\mathit{fixcomp}$, the result would not reflect these extensions (i.e. the services $resize$ and $selectAll$ would not be available). A more generic composition abstraction would use polymorphic form extension:

$$
\mathbf{def}\ \mathit{compose}(L,R,r) \quad = \quad (\nu\ d)(\mathit{dispatch}(L,R,d) \mid d(C).\overline{r}(LR\langle \mathit{close}=C_{\mathit{close}}\rangle))
$$

Given the original GUIList and Multiselector components, the $compose$ abstraction returns exactly the same composite as the old version. However, due to the usage of

[2]We use similar derived forms as for the mini π-calculus.

polymorphic form extension, the resulting composite also reflects extensions of the argument components like $resize$ or $selectAll$. The $compose$ abstraction is more generic than $fixcomp$ as it only assumes that both arguments offer a $close$ service. Note that if both arguments offer other services with the same name, only that of the right-hand side argument will be available in the composite component.

In the following, we will not discuss the full definition of the $\pi\mathcal{L}$-calculus, as it is similar to the FORM calculus introduced in the next section. For a detailed discussion about the $\pi\mathcal{L}$-calculus, refer to [Lum99].

7.3 The FORM calculus

As we will discuss in Chapters 8 and 9, the $\pi\mathcal{L}$-calculus is a suitable formal foundation for modelling objects and compositional abstractions, and has several advantages over the plain π-calculus. In particular, our experiments have revealed that polymorphic form extension is a powerful mechanism for software composition. However, there are situations where the expressive power of the $\pi\mathcal{L}$-calculus is not enough. As an example, consider the encoding of the encapsulation operator proposed by Van Limberghen and Mens [VLM96], where it is necessary to *remove* a set of labels from a form, which cannot be directly expressed in the $\pi\mathcal{L}$-calculus. In this section, we present the FORM calculus, an extension of the $\pi\mathcal{L}$-calculus with additional operators on forms and agent expressions.

More precisely, the FORM calculus extends the syntax for forms of the $\pi\mathcal{L}$-calculus with the notion of *polymorphic form restriction* which removes the set of labels defined in a form variable from a given form. Note that polymorphic form restriction can be considered as the "inverse" operation to polymorphic form extension.

Furthermore, the FORM calculus defines the concept of *label matching*, a mechanism to check for the name bound by a label l in a given form F (written $[F \leftarrow l]$), which is similar to *name matching* in the π-calculus [MPW92]. We define label matching in a way that a match is successful if i) a form defines a binding for the given label and ii) the name bound by the label is not equal to the empty binding $\mathcal{E}$. This decision is motivated by our goal of defining a composition calculus: label matching should allow us to check whether a component whose interface is given by a form offers (or does not offer) a specific service. The current definition of label matching ensures that if a match is successful, then the name bound by the given label denotes a valid entry point to the corresponding service. As we will illustrate in section 7.3.7, an empty binding cannot be used for further communications.

Parrow and Sangiorgi have shown that if both matching and mismatching on primitives of the calculus (such as forms and labels in the FORM calculus) are necessary, it is not possible to express such a behaviour with only one alternative [PS95]. However, as we will show in chapter 9, there are contexts where a negative match is needed. Therefore, we have extended the syntax of label matching and allow *two* continuation agents (instead of only one): $[F \leftarrow l]\ A_1\ A_2$ yields A_1 for a successful match (i.e. the form F defines a nonempty binding for the label l), A_2 if the match fails.

An alternative to label matching is to define matching on names (like in the π-calculus): $[F \leftarrow l]$ can be expressed as $[F_l \neq \mathcal{E}]$. This implies that any expression using label matching can be translated into an equivalent expression only using name matching, but not vice versa. Exchanging label matching with name matching in the FORM calculus would lead to a calculus with more expressive power. However, our experience with modelling objects and compositional abstractions in various flavours of the π-calculus never revealed the necessity to test for name equality. In addition, the encoding of the FORM calculus in the mini π-calculus presented in section 7.3.11 cannot be defined as a simple adaptation of the corresponding translation of the $\pi\mathcal{L}$-calculus into the mini π-calculus.

In this section, we formally introduce the FORM calculus, define an observable equivalence based on a weak bisimulation, and show that the FORM calculus can be faithfully encoded in the π-calculus and, therefore, also in the $\pi\mathcal{L}$-calculus.

7.3.1 Names and forms

The most primitive entity, as in the mini π-calculus, is a *name*. However, unlike in the π-calculus, names are only used as the *subject* of a communication. The role of the *object* of a communication is taken by so-called *forms*. Forms are finite mappings from an infinite set $\mathcal{L}$ of labels to an infinite set $\mathcal{N}^+ = \mathcal{N} \cup \{\mathcal{E}\}$, the set of names $\mathcal{N}$ extended by $\mathcal{E}$, denoting the empty binding. In the following, a, b, c, d range over the set $\mathcal{N}$ of names, x, y, z range over $\mathcal{N}^+$, F, G, H, I range over forms, X, Y, Z range over form variables, and l, m, n range over the set of labels $\mathcal{L}$. The syntax for forms is defined as follows:

$$
\begin{array}{rcll}
F & ::= & \langle\rangle & \textit{empty form} \\
& | & F\langle l{=}V\rangle & \textit{binding extension} \\
& | & X & \textit{form variable} \\
& | & F{\cdot}X & \textit{polymorphic extension} \\
& | & F\backslash X & \textit{polymorphic restriction}
\end{array}
$$

where

$$
\begin{array}{rcll}
V & ::= & \mathcal{E} & \textit{empty binding} \\
& | & a & \textit{simple name} \\
& | & X_l & \textit{projection}
\end{array}
$$

In form expressions, binding extension has precedence over polymorphic form extension, which in turn has precedence over polymorphic restriction. Parenthesis may be used to group form expressions in order to overcome the default precedence rules.

Form variables and *projections* deserve a special attention. In the π-calculus we only have names. A name that occurs as object in an input prefix, for example the name y in $x(y).P$, is said to be the location of the place where an actually received value z will go in process P (i.e. in process P name y will be *instantiated* by name z).

In the FORM calculus, however, form variables are used as the object part of an input prefix. These form variables can be seen as *polymorphic placeholders* for forms and, in contrast to the π-calculus, form variables are values and not references. Therefore, form variables cannot be instantiated by the received form; they are simply substituted by the received form.

On the other hand, *name projections* denote locations of names in the FORM calculus. In fact, projections are *named formal process parameters* which can be distributed over terms. A projection X_l has to be read as selection of the parameter named by l. Moreover, if X_l occurs in an agent $a(X).A$, then X_l will be instantiated by z if X_l maps to z.

As a consequence, unlike in the π-calculus, where *name instantiation* (or the substitution of names to names) is done in one step, the FORM calculus requires two steps: a first step in order to substitute all form variables X in agent A for some received form value F, and a second step to instantiate (or substitute) all projections X_l in A to the name denoted by X_l. Both substitutions are treated as one atomic action. Therefore, in the FORM calculus, we say that a term is instantiated rather than the names of a term are instantiated.

Before we can define name projection, we need to define the notion of the *variables of a form* and *closed forms*, as name projection is only defined for closed forms. We also define a congruence over closed forms, introduce *binding restriction* as a special case of binding extension, and prove structural properties of both polymorphic form extension and restriction. Furthermore, we define the notion of a *normalized form* for closed forms and show that any closed form F can be replaced by an equivalent normalized form F'. Finally, we introduce the notions of *names* and *labels* of forms, respectively.

Definition 7.1 (Variables of a form) *The set of variables of a form F, written $\mathcal{V}(F)$, is defined as:*

$$\begin{array}{rcl}
\mathcal{V}(\langle\rangle) & = & \emptyset \\
\mathcal{V}(F\langle l{=}x\rangle) & = & \mathcal{V}(F) \\
\mathcal{V}(X) & = & \{X\} \\
\mathcal{V}(F\langle l{=}X_l\rangle) & = & \{X\} \cup \mathcal{V}(F) \\
\mathcal{V}(F{\cdot}X) & = & \{X\} \cup \mathcal{V}(F) \\
\mathcal{V}(F\backslash X) & = & \{X\} \cup \mathcal{V}(F)
\end{array}$$

Definition 7.2 (Closed forms) *A form F is closed if it does not contain any form variable: $\mathcal{V}(F) = \emptyset$.*

Definition 7.3 (Name projection) *Given two closed forms F and G and a form H substituting form variable X, then the application of a label $l \in \mathcal{L}$ to form H (mapping from $\mathcal{L}$ to $\mathcal{N}^+$), written H_l, is called name projection and is recursively defined as follows:*

$$\begin{array}{rcll}
\langle\rangle_l & = & \mathcal{E} & \\
(F\langle l{=}x\rangle)_l & = & x & \\
(F\langle m{=}y\rangle)_l & = & F_l & \textit{if } m \neq l \\
(F \cdot G)_l & = & \left\{ \begin{array}{ll} F_l, & \textit{if } G_l = \mathcal{E} \\ G_l, & \textit{otherwise} \end{array} \right. & \\
(F\backslash G)_l & = & \left\{ \begin{array}{ll} F_l, & \textit{if } G_l = \mathcal{E} \\ \mathcal{E} & \textit{otherwise} \end{array} \right. &
\end{array}$$

Note that a form may have multiple bindings for label l. In this case, Definition 7.3 ensures that a name projection always extracts the *rightmost binding*. This allows an agent to override a binding with a new one, preserving all other bindings.

Definition 7.4 (Equivalence of forms) *Two closed forms F and G are equivalent, written $F \equiv G$, if and only if for all $l \in \mathcal{L}$ it holds that:*

$$F_l \ = \ G_l$$

The reader should note that, by definition, if $F \equiv G$, then for all $l \in \mathcal{L}$ it holds that $F_l = G_l$ (i.e. $F \equiv G \ \Leftrightarrow \ F_l = G_l \ \ \forall\, l \in \mathcal{L}$).

Proposition 7.1 *$\equiv$ is a congruence relation for closed forms.*

PROOF: In order to prove that $\equiv$ is a congruence relation, we have to prove that it is an equivalence relation and that congruence is preserved under all operators for forms. In the following, we assume that F, G, and H are closed forms and $F \equiv G$.

- $\equiv$ being an equivalence relation immediately follows from Definition 7.3.
- $F\langle l{=}V\rangle \equiv G\langle l{=}V\rangle$:
  By definition, it holds that $F_l = V$ and $G_l = V$ (for any value V). Furthermore, for any label $m \in \mathcal{L}{-}\{l\}$ it holds by assumption that $F_m = G_m$. Hence, $(F\langle l{=}V\rangle)_n = (G\langle l{=}V\rangle)_n$ for all $n \in \mathcal{L}$, and it immediately follows that $F\langle l{=}V\rangle \equiv G\langle l{=}V\rangle$.
- $F \cdot H \equiv G \cdot H$:
  Consider $\mathcal{L}_H = \{n | n \in \mathcal{L} \wedge H_n \neq \mathcal{E}\}$. By definition, it holds that for all $l \in \mathcal{L}_H$ $(F{\cdot}H)_l = H_l$ and $(G{\cdot}H)_l = H_l$. Furthermore, for all $m \in \mathcal{L} - \mathcal{L}_H$ it holds that $(F{\cdot}H)_m = F_m$ and $(G{\cdot}H)_m = G_m$. Hence, it immediately follows that $F{\cdot}H \equiv G{\cdot}H$.
- $F \setminus H \equiv G \setminus H$:
  Consider $\mathcal{L}_H = \{n | n \in \mathcal{L} \wedge H_n \neq \mathcal{E}\}$. By definition, it holds that for all $l \in \mathcal{L}_H$ $(F\backslash H)_l = \mathcal{E}$ and $(G\backslash H)_l = \mathcal{E}$. Furthermore, for all $m \in \mathcal{L} - \mathcal{L}_H$ it holds that $(F\backslash H)_m = F_m$ and $(G\backslash H)_m = G_m$. Hence, it immediately follows that $F\backslash H \equiv G\backslash H$.

The proofs for $H{\cdot}F \equiv H{\cdot}G$ and $H\backslash F \equiv H\backslash G$ are similar to the proofs for $F{\cdot}H \equiv G{\cdot}H$ and $F\backslash H \equiv G\backslash H$, respectively. □

Proposition 7.2 *Given a closed form F, it holds that:*

$$\begin{array}{rcll} \langle\rangle & \equiv & \langle\rangle & \\ F\langle l{=}x\rangle\langle m{=}y\rangle & \equiv & F\langle m{=}y\rangle\langle l{=}x\rangle & \textit{if } m \neq l \\ F\langle l{=}x\rangle\langle l{=}y\rangle & \equiv & F\langle l{=}y\rangle & \end{array}$$

PROOF: In the following, we assume that F is a closed form and $x, y \in \mathcal{N}^+$.

- $\langle\rangle \equiv \langle\rangle$ is vaconsly true.
- $F\langle l{=}x\rangle\langle m{=}y\rangle \equiv F\langle m{=}y\rangle\langle l{=}x\rangle$:
  Consider $F' = F\langle l = x\rangle\langle m = y\rangle$ and $F'' = F\langle m = y\rangle\langle l = x\rangle$. By definition, it holds that $F'_l = x$, $F''_l = x$, $F'_m = y$, and $F''_m = y$. Furthermore, for any label $n \in \mathcal{L} - \{l, m\}$ it holds that $F'_n = F_n$ and $F''_n = F_n$, which implies that $F' \equiv F''$ as required.
- $F\langle l{=}x\rangle\langle l{=}y\rangle \equiv F\langle l{=}y\rangle$:
  Consider $F' = F\langle l = x\rangle\langle l = y\rangle$ and $F'' = F\langle l = y\rangle$. By definition, it holds that $F'_l = y$ and $F''_l = y$. Furthermore, for any label $n \in \mathcal{L} - \{l\}$ it holds that $F'_n = F_n$ and $F''_n = F_n$, which implies that $F' \equiv F''$ as required. □

The extension of a form F with an empty binding for a label l (i.e. $F' = F\langle l{=}\mathcal{E}\rangle$) deserves special attention. From a different perspective, F' can be considered as a form where all bindings for label l have been removed. Since this behaviour is of fundamental importance for the FORM calculus (and allows us to interpret polymorphic form restriction in a more natural way), we define *binding restriction* a special notation for extension with an empty binding.

Definition 7.5 (Binding restriction) *Given a form F, then the application of a binding restriction with the label l, written $F\backslash l$, is defined as follows:*

$$F\backslash l \quad = \quad F\langle l{=}\mathcal{E}\rangle$$

Proposition 7.3 *Given a closed form F, it holds that:*

$$\begin{array}{rcll} \langle\rangle\backslash l & \equiv & \langle\rangle & \\ (F\langle l{=}x\rangle)\backslash l & \equiv & F\backslash l & \\ (F\langle m{=}y\rangle)\backslash l & \equiv & (F\backslash l)\langle m{=}y\rangle & \textit{if } m \neq l \end{array}$$

PROOF: The result immediately follows from Definition 7.3 and Proposition 7.1. □

The idea behind binding restriction is to *recursively* remove all bindings of a label for a given form. Proposition 7.3 illustrates how a form containing extensions with an empty binding can be replaced by an equivalent form which does not contain an empty binding. Furthermore, the definition of binding restriction ensures that $F\backslash l$ is always well-formed, even if F does not define a binding for l. Finally, $(F\langle l{=}x\rangle)\backslash l$ is in general not equal to F (the form F may contain other bindings for the label l).

Proposition 7.4 *Given two closed forms F and G, it holds that:*

$$\begin{array}{rcl} F \cdot \langle\rangle & \equiv & F \\ \langle\rangle \cdot F & \equiv & F \\ F \cdot G\langle l{=}a\rangle & \equiv & (F \cdot G)\langle l{=}a\rangle \\ F \cdot G\langle l{=}\mathcal{E}\rangle & \equiv & (F \cdot G)\langle l{=}F_l\rangle \end{array}$$

PROOF: In the following, we assume that F and G are closed forms and $a \in \mathcal{N}$.

- The correctness of $F\langle\rangle \equiv F$ and $\langle\rangle F \equiv F$ immediately follows from Definition 7.3.
- $F \cdot G\langle l = a\rangle \equiv (F \cdot G)\langle l = a\rangle$:
Consider $F' = F{\cdot}G\langle l = a\rangle$ and $F'' = (F{\cdot}G)\langle l = a\rangle$. By definition, it holds that $F'_l = a$ and $F''_l = a$. Furthermore, for any label $n \in \mathcal{L} - \{l\}$ it holds that $F'_n = F_n$ if $G_n = \mathcal{E}$, $F'_n = G_n$ otherwise. Similar for F''_n. Hence, $F' \equiv F''$ as required.
- $F \cdot G\langle l = \mathcal{E}\rangle \equiv (F \cdot G)\langle l = F_l\rangle$:
Consider $F' = F{\cdot}G\langle l = \mathcal{E}\rangle$ and $F'' = (F{\cdot}G)\langle l = F_l\rangle$. By definition, it holds that $F'_l = F_l$ (since $(G\langle l = \mathcal{E}\rangle)_l = \mathcal{E}$) and $F''_l = F_l$. Furthermore, for any label $n \in \mathcal{L} - \{l\}$ it holds that $F'_n = F_n$ if $G_n = \mathcal{E}$, $F'_n = G_n$ otherwise. Similar for F''_n. Hence, $F' \equiv F''$ as required. □

Proposition 7.4 illustrates that name projection is defined in a way that in the context of polymorphically extending form F with form G, all *nonempty* label bindings of G are added to the bindings of F and that in case of multiple bindings of the same label, the rightmost binding is extracted in a name projection. Furthermore, it can be easily shown that for a given form F, $F\langle l = \mathcal{E}\rangle$ is not equal to $F\langle\rangle\langle l = \mathcal{E}\rangle$: in the former case, projecting label l yields $\mathcal{E}$ whereas in the latter case, the projection results in F_l.

Proposition 7.5 *Given two closed forms F and G, it holds that:*

$$\begin{array}{rcl} F \setminus \langle\rangle & \equiv & F \\ \langle\rangle \setminus F & \equiv & \langle\rangle \\ F \setminus G\langle l = a\rangle & \equiv & F\langle l = \mathcal{E}\rangle \setminus G \\ F \setminus G\langle l = \mathcal{E}\rangle & \equiv & (F \setminus G)\langle l = F_l\rangle \end{array}$$

PROOF: The proof is similar to the one for Proposition 7.4 and immediately follows from Definition 7.3 and Proposition 7.1. □

Proposition 7.5 illustrates that only those labels bound in G are removed from F which bind a (valid) name (and not $\mathcal{E}$). This is also a consequence of how we previously illustrated label matching (refer also to section 7.3.7). Furthermore, the definition of name projection ensures that if G_l is equal to $\mathcal{E}$, then $(F \setminus G)_l$ yields F_l. On the other hand, if we changed Definition 7.3 in a way that $F \setminus G\langle l = x\rangle$ yields $(F \setminus l) \setminus G$ for arbitrary $x \in \mathcal{N}^+$, then $(F \setminus G\langle l = x\rangle)_l$ always yields the empty binding which, however, is rather contra-intuitive. Note that, similar to binding restriction, $(F{\cdot}G) \setminus G$ is generally not equal to F.

Definition 7.6 (Normalized forms) *A closed form F is normalized if it is defined as:*

$$F = \begin{cases} \langle\rangle\langle l_1 = b_1\rangle \ldots \langle l_n = b_n\rangle, & for\ n \geq 1,\ b_1, \ldots, b_n \in \mathcal{N} \\ \langle\rangle, & for\ n = 0 \end{cases}$$

with pairwise distinct labels l_i.

Throughout the rest of this work, we use the notation $\langle\widetilde{l=b}\rangle$ to denote a closed and normalized form.

Theorem 7.1 *For any closed form F, there exists an equivalent form F' (i.e. $F' \equiv F$) which is normalized.*

PROOF: We proceed by induction over the structure of F. Since $F = \langle\rangle$ is already normalized, we will only consider forms which contain at least a binding extension, polymorphic extension, or polymorphic restriction.

1. Using Propositions 7.4 and 7.5, it is possible to remove any application of polymorphic extension or restriction from a form F and find an equivalent form F' which only contains binding extensions.
2. Using Proposition 7.3 and the fact that $\equiv$ is a congruence relation, it is possible to remove all bindings of the form $\langle l = \mathcal{E}\rangle$ (and the corresponding overridden bindings) from F' and define a form $F'' \equiv F'$ which only contains "positive" bindings (i.e. bindings of the form $\langle l_i = b_i\rangle$).
3. Finally, using Proposition 7.2, it is possible to remove all multiple bindings and make all labels pairwise distinct. □

Note that steps 1 and 2 may reduce a form to the empty form.

Definition 7.7 (Names of a form) *The set of names of a form F, written $\mathcal{N}(F)$, is defined as:*

$$
\begin{array}{rcl}
\mathcal{N}(\langle\rangle) & = & \emptyset \\
\mathcal{N}(F\langle l=a\rangle) & = & \{a\} \cup \mathcal{N}(F) \\
\mathcal{N}(F\langle l=X_k\rangle) & = & \mathcal{N}(F) \\
\mathcal{N}(F{\cdot}X) & = & \mathcal{N}(F) \\
\mathcal{N}(F\backslash X) & = & \mathcal{N}(F)
\end{array}
$$

The reader should note the set of names of a form includes all bound names, even the names that cannot be accessed using name projection (i.e. the set of names of the form $F = \langle\rangle\langle l=a\rangle\langle l=b\rangle\langle l=c\rangle$ is equal to $\{a, b, c\}$).

Definition 7.8 (Labels of a form) *The set of labels of a form F, written $\mathcal{L}(F)$, is defined as:*

$$
\begin{array}{rcl}
\mathcal{L}(\langle\rangle) & = & \emptyset \\
\mathcal{L}(F\langle l=x\rangle) & = & \{l\} \cup \mathcal{L}(F) \\
\mathcal{L}(F\langle l=X_k\rangle) & = & \{l\} \cup \mathcal{L}(F) \\
\mathcal{L}(F{\cdot}X) & = & \mathcal{L}(F) \\
\mathcal{L}(F\backslash X) & = & \mathcal{L}(F)
\end{array}
$$

7.3.2 The language

The class $\mathcal{A}$ of FORM calculus agents is built using the operators of inaction, parallel composition, restriction, input prefix, replication, output, and matching. A, B, C to range over the class of agents. The syntax for agents is defined as follows:

$$
\begin{array}{rcll}
A & ::= & \mathbf{0} & \textit{inactive agent} \\
& | & A \mid A & \textit{parallel composition} \\
& | & (\nu\ a)A & \textit{restriction} \\
& | & V(X).A & \textit{input}\ \ (\textit{receive form in}\ X) \\
& | & !V(X).A & \textit{replication} \\
& | & \overline{V}(F) & \textit{output}\ \ (\textit{send form}\ F) \\
& | & [F \leftarrow l]\ A\ A & \textit{matching}
\end{array}
$$

0 denotes the inactive agent. Parallel composition runs two agents in parallel. The restriction $(\nu\ a)A$ makes the name a local to the agent A (i.e. it creates a *fresh* name a with scope A).[3] An input-prefixed agent $V(X).A$ waits for a form F to be sent along the channel denoted by value V and then behaves like $A\{F/X\}$, where $\{F/X\}$ is the substitution of all form variables X with form F. An output $\overline{V}(F)$ emits a form F along the channel denoted by value V. A replication $!V(X).A$ stands for a countably infinite number of copies of $V(X).A$ in parallel. Like in the mini π-calculus, replication is only defined for input-prefixed agents. A matching $[F \leftarrow l]\ A_1\ A_2$ yields A_1 if the match succeeds (i.e. the form F defines a nonempty binding for label l), A_2 otherwise. The reader should note that matching is only defined on closed forms. Finally, input prefix, restriction, and replication have precedence over matching, which in turns has precedence over parallel composition.

Definition 7.9 (Labels of an agent) *The set of labels of an agent A, written $\mathcal{L}(A)$, is defined as:*

$$
\begin{array}{rcl}
\mathcal{L}(\mathbf{0}) & = & \emptyset \\
\mathcal{L}(A_1 \mid A_2) & = & \mathcal{L}(A_1) \cup \mathcal{L}(A_2) \\
\mathcal{L}((\nu\ a)A) & = & \mathcal{L}(A) \\
\mathcal{L}(a(X).A) & = & \mathcal{L}(A) \\
\mathcal{L}(Y_l(X).A) & = & \{l\} \cup \mathcal{L}(A) \\
\mathcal{L}(\overline{a}(F)) & = & \mathcal{L}(F) \\
\mathcal{L}(\overline{Y_l}(F)) & = & \{l\} \cup \mathcal{L}(F) \\
\mathcal{L}([F \leftarrow l]\ A_1\ A_2) & = & \{l\} \cup \mathcal{L}(F) \cup \mathcal{L}(A_1) \cup \mathcal{L}(A_2)
\end{array}
$$

7.3.3 Encoding of booleans and choice

The following two examples present the encoding of booleans and input-guarded choice in the FORM calculus, respectively. The examples show that the FORM calculus is a compact formalism for encoding these kind of abstractions.

[3] A local channel name can be communicated to other agents. This mechanism is called *scope extrusion*.

Unlike the π-calculus and the $\pi\mathcal{L}$-calculus, where the boolean values true and false have to be encoded as *processes*, the encoding in the FORM calculus allows a different approach:

$$\begin{array}{rcl} \mathit{True} & = & \langle \mathit{true}{=}\_\rangle \\ \mathit{False} & = & \langle \mathit{false}{=}\_\rangle \end{array}$$

Both boolean values True and False are encoded as forms defining a binding for the labels true and false, respectively. The abbreviations $\langle \mathit{true}{=}\_\rangle$ and $\langle \mathit{false}{=}\_\rangle$ are used to denote a binding to an arbitrary name (not equal to $\mathcal{E}$). Note that the name bound to the labels true and false is not of importance (and is not accessed in any boolean operation).

Negation, disjunction, and conjunction of the boolean encoding presented above can be defined as follows:

$$\begin{array}{rcl} \mathbf{def}\ \mathit{Not}(B, r) & = & \overline{r}(\langle \mathit{true}{=}\_\rangle\langle \mathit{false}{=}\_\rangle\backslash B) \\ \mathbf{def}\ \mathit{And}(B_1, B_2, r) & = & [B_1{\leftarrow}\mathit{false}]\ \overline{r}(B_1)\ \overline{r}(B_2) \\ \mathbf{def}\ \mathit{Or}(B_1, B_2, r) & = & [B_1{\leftarrow}\mathit{true}]\ \overline{r}(B_1)\ \overline{r}(B_2) \end{array}$$

Unlike the mini π-process Not discussed in section 7.1, the corresponding FORM calculus agent does not have to perform a test on the boolean value; polymorphic form restriction is used to reverse the binding of the labels true and false, respectively. The agents for And and Or only perform a test on the first argument and return one of the two arguments as the result.

Using this encoding of booleans, an "$\mathrm{if\ then\ else}$" construct can be most naturally encoded as

$$[\![\mathrm{if}\ b\ \mathrm{then}\ P\ \mathrm{else}\ Q]\!] \quad = \quad [b{\leftarrow}\mathit{true}]\ \ P\ \ Q$$

The "original" version of the π-calculus defines the notion of *summation* over processes, which is often referred to as *choice* (i.e. a process $P \equiv P_1 + P_2$ behaves either as P_1 or P_2) [MPW92]. In the context of an asynchronous calculus, the special case of *input-guarded choice* is often used to determine the course of computation based on the value of a preceding input:

$$A \ \equiv\ \sum_{j \in J} a_j(X).A_j$$

Nestmann and Pierce have defined an encoding of an asynchronous π-calculus with input-guarded choice into an asynchronous π-calculus without choice [NP96]. Using the encodings of booleans defined above, this encoding can be easily adapted to the FORM calculus:

$$[\![A]\!] \ \equiv\ (\nu\ c)(\overline{c}(\mathit{True}) \mid \prod_{j \in J} \mathrm{Branch}(a_j(x).A_j))$$

with

$$\text{Branch}(a_j(x).A_j) \;=\; a_j(X).c(B).[B \leftarrow true]\, [\![A_j]\!]\, (\overline{a_j}(X) \mid \overline{c}(False))$$

For each choice in the agent A given above, the translation runs a mutual exclusion protocol, by installing a local channel c (acting as a lock) in the scope of the parallel composition of its branches. The branches concurrently try to acquire the lock after reading messages from the environment. Only the first branch managing to interrogate the lock will proceed with its continuation (i.e. it will be the only branch reading the value $True$ from the lock) and, therefore, commit the choice. Every other branch will then be forced to resend its message to the environment and abort its continuation. The resending of messages by non-chosen branches essentially reflects the asynchronous character of the encoding.

7.3.4 Name binding

Both the input prefix and the restriction operator are binders for names in the π-calculus. In the FORM calculus, however, only the operator $(\nu\ a)A$ acts as a binder for names occurring free in an agent whereas the input prefix $V(X)$ is the binding operator for form variables. $\text{fn}(A)$ and $\text{bn}(A)$ denote the set of *free* and *bound names* of an agent and $\text{fv}(A)$ and $\text{bv}(A)$ to denote the set of *free* and *bound form variables* of an agent, respectively.

Definition 7.10

a) *The set of free names of an agent A, written* $\text{fn}(A)$*, is inductively given by:*

$$\begin{aligned}
\text{fn}(\mathbf{0}) &= \emptyset, \\
\text{fn}(A_1 \mid A_2) &= \text{fn}(A_1) \cup \text{fn}(A_2), \\
\text{fn}((\nu\ a)A) &= \text{fn}(A) - \{a\}, \\
\text{fn}(a(X).A) = \text{fn}(!a(X).A) &= \{a\} \cup \text{fn}(A), \\
\text{fn}(Y_l(X).A) = \text{fn}(!Y_l(X).A) &= \text{fn}(A) \\
\text{fn}(\overline{a}(F)) &= \{a\} \cup \mathcal{N}(F), \\
\text{fn}(\overline{Y_l}(F)) &= \mathcal{N}(F), \\
\text{fn}([F \leftarrow l]\ A_1\ A_2) &= \mathcal{N}(F) \cup \text{fn}(A_1) \cup \text{fn}(A_2).
\end{aligned}$$

b) *The set of bound names of an agent A, written* $\text{bn}(A)$*, is inductively given by:*

$$\begin{aligned}
\text{bn}(\mathbf{0}) &= \emptyset, \\
\text{bn}(A_1 \mid A_2) &= \text{bn}(A_1) \cup \text{bn}(A_2), \\
\text{bn}((\nu\ a)A) &= \{a\} \cup \text{bn}(A), \\
\text{bn}(V(X).A) = \text{bn}(!V(X).A) &= \text{bn}(A), \\
\text{bn}(\overline{V}(F)) &= \emptyset, \\
\text{bn}([F \leftarrow l]\ A_1\ A_2) &= \text{bn}(A_1) \cup \text{bn}(A_2).
\end{aligned}$$

c) *The set of names of an agent A, written* $\text{n}(A)$*, is given by* $\text{n}(A) = \text{fn}(A) \cup \text{bn}(A)$.

d) The set of free variables of an agent A, written $\mathrm{fv}(A)$*, is inductively given by:*

$$\begin{array}{rcl}
\mathrm{fv}(\mathbf{0}) & = & \emptyset, \\
\mathrm{fv}(A_1 \mid A_2) & = & \mathrm{fv}(A_1) \cup \mathrm{fv}(A_2), \\
\mathrm{fv}((\nu\ a)A) & = & \mathrm{fv}(A), \\
\mathrm{fv}(!a(X).A) = \mathrm{fv}(a(X).A) & = & \mathrm{fv}(A) - \{X\}, \\
\mathrm{fv}(!Y_l(X).A) = \mathrm{fv}(Y_l(X).A) & = & (\{Y\} \cup \mathrm{fv}(A)) - \{X\}, \\
\mathrm{fv}(\overline{a}(F)) & = & \mathcal{V}(F), \\
\mathrm{fv}(\overline{Y_l}(F)) & = & \{Y\} \cup \mathcal{V}(F), \\
\mathrm{fv}([F \leftarrow l]\ A_1\ A_2) & = & \mathcal{V}(F) \cup \mathrm{fv}(A_1) \cup \mathrm{fv}(A_2).
\end{array}$$

e) The set of bound variables of an agent A, written $\mathrm{bv}(A)$*, is inductively given by:*

$$\begin{array}{rcl}
\mathrm{bv}(A_1 \mid A_2) & = & \mathrm{bv}(A_1) \cup \mathrm{bv}(A_2), \\
\mathrm{bv}((\nu\ a)A) & = & \mathrm{bv}(A), \\
\mathrm{bv}(V(X).A) = \mathrm{bv}(!V(X).A) & = & \{X\} \cup \mathrm{bv}(A), \\
\mathrm{bv}(\mathbf{0}) = \mathrm{bv}(\overline{V}(F)) & = & \emptyset, \\
\mathrm{bv}([F \leftarrow l]\ A_1\ A_2) & = & \mathrm{bv}(A_1) \cup \mathrm{bv}(A_2).
\end{array}$$

f) The set of variables of an agent A, written $\mathrm{v}(A)$*, is given by* $\mathrm{v}(A) = \mathrm{fv}(A) \cup \mathrm{bv}(A)$.

Definition 7.11 (Closed agents) *An agent A is closed if it does not contain any free form variables (i.e.* $\mathrm{fv}(A) = \emptyset$*).*

7.3.5 Form substitution

Like in the $\pi\mathcal{L}$-calculus, $A\{F/X\}$ denotes the substitution of all free occurrences of form variable X with form F in agent A. In the following, σ ranges over form substitutions, and substitutions have precedence over the operators of the language.

Definition 7.12 (Form substitution) *Let $\sigma = \{F/X\}$ and F be closed. Then the effect of the substitution σ on the agent A, written $A\sigma$, is defined inductively below. In order to avoid that free names of F become accidentally bound in $A\sigma$ (underneath a restriction operator), we assume that the conflicting names in A have been previously α-converted to fresh names (i.e.* $\mathrm{bn}(A) \cap \mathcal{N}(F) = \emptyset$*).*

$$\begin{array}{rcl}
\mathbf{0}\sigma & = & \mathbf{0} \\
(A_1 \mid A_2)\sigma & = & (A_1\sigma) \mid (A_2\sigma) \\
((\nu\ a)A)\sigma & = & (\nu\ a)(A\sigma) \\
(V(X).A)\sigma & = & (V\sigma)(X).A \\
(V(Y).A)\sigma & = & (V\sigma)(Y).(A\sigma), \quad \textit{if } Y \neq X
\end{array}$$

$$
\begin{array}{rcl}
(!V(X).A)\sigma & = & !(V(X).A)\sigma \\
(\overline{V}(G))\sigma & = & \overline{(V\sigma)}(G\sigma) \\
([G{\leftarrow}l]\ A_1\ A_2)\sigma & = & ([(G\sigma){\leftarrow}l])\ (A_1\sigma)\ (A_2\sigma)
\end{array}
$$

with

$$
V\sigma = \left\{ \begin{array}{ll} F_l, & \textit{if } V = X_l \\ V, & \textit{otherwise} \end{array} \right.
$$

$$
G\sigma = \left\{ \begin{array}{ll} \langle\rangle, & \textit{if } G = \langle\rangle \\ (H\sigma)\langle l{=}V\sigma\rangle, & \textit{if } G = H\langle l{=}V\rangle \\ F, & \textit{if } G = X \\ (H\sigma){\cdot}(I\sigma), & \textit{if } G = H{\cdot}I \\ (H\sigma)\backslash(I\sigma), & \textit{if } G = H\backslash I \end{array} \right.
$$

By definition, F must be closed in $A\{F/X\}$. Therefore, each time the substitution yields a projection F_l, the projection is immediately replaced by the result of F_l (i.e. a simple name or $\mathcal{E}$). A form substitution $A\sigma$ simultaneously substitutes all free occurrences of form variable X by form value F in A, and all projections X_l in A by x if X_l maps x. Using this approach, it is possible to avoid a "trigger" operation that instantiates, when needed, each X_l in A to its corresponding name (for example, if X_l occurs as the subject of an outermost input prefix or output particle). The same applies for substituting all free occurrences of a form variable X in a matching agent.

7.3.6 α-substitution

Unlike in the π-calculus, there is no general *name substitution* in the FORM calculus. Therefore, it is necessary to define explicitly *alpha-conversion* of bound names and bound variables, respectively. The alpha-substitution of bound names in agent A is written as $A\{\tilde{c}/\tilde{a}\}^{\mathcal{N}}_{\alpha}$ whereas $A\{Y/X\}^{\mathcal{V}}_{\alpha}$ denotes the alpha-substitution of bound variables in A.[4] In the following, $\alpha^{\mathcal{N}}$ or $\alpha^{\mathcal{V}}$ range over alpha-substitutions. Like form substitution, alpha-substitutions have precedence over the operators of the language. If $\alpha^{\mathcal{N}} = \{\tilde{c}/\tilde{a}\}^{\mathcal{N}}_{\alpha}$ ($\tilde{a} = a_1, ..., a_n; \tilde{c} = c_1, ..., c_n$) and $\alpha^{\mathcal{V}} = \{Y/X\}^{\mathcal{V}}_{\alpha}$, then $A\alpha^{\mathcal{N}/\mathcal{V}}$ is the agent obtained from A by replacing the a_i's with the c_i's and all occurrences of X with Y, respectively.

Definition 7.13 (α-substitution) *Let $\alpha^{\mathcal{N}} = \{\tilde{c}/\tilde{a}\}^{\mathcal{N}}_{\alpha}$ and $\alpha^{\mathcal{V}} = !\{Y/X\}^{\mathcal{V}}_{\alpha}$. The effect of the substitution $\alpha^{\mathcal{N}/\mathcal{V}}$ on the agent A, written $A\alpha^{\mathcal{N}/\mathcal{V}}$, is defined inductively as follows:*

$$
\begin{array}{rcl}
(A_1 \mid A_2)\alpha^{\mathcal{N}/\mathcal{V}} & = & (A_1\alpha^{\mathcal{N}/\mathcal{V}}) \mid (A_2\alpha^{\mathcal{N}/\mathcal{V}}) \\
((\nu\ b)A)\alpha^{\mathcal{N}} & = & (\nu\ b')(A\alpha^{\mathcal{N}}),\ \ b' = \left\{ \begin{array}{ll} b, & \textit{if } b \notin \tilde{a} \\ c_i, & \textit{if } b = a_i \textit{ and } a_i\alpha^{\mathcal{N}} = c_i \end{array} \right. \\
((\nu\ b)A)\alpha^{\mathcal{V}} & = & (\nu\ b)(A\alpha^{\mathcal{V}}) \\
(V(X).A)\alpha^{\mathcal{N}} & = & (V\alpha^{\mathcal{N}})(X).A\alpha^{\mathcal{N}}
\end{array}
$$

[4] Like in the π-calculus, $\tilde{c}$ is equal to $c_1, \ldots, c_n$ for a given n.

$$
\begin{aligned}
(V(Z).A)\alpha^{\mathcal{V}} &= (V\alpha^{\mathcal{V}})(Z').A\alpha^{\mathcal{V}},\;\; Z' = \begin{cases} Z, & \mathit{if}\ Z \neq X \\ Y, & \mathit{if}\ Z = X,\ \mathit{and}\ X\alpha^{\mathcal{V}} = Y \end{cases} \\
(!V(X).A)\alpha^{\mathcal{N}/\mathcal{V}} &= !(V(X).A)\alpha^{\mathcal{N}/\mathcal{V}} \\
(\overline{V}(F))\alpha^{\mathcal{N}/\mathcal{V}} &= \overline{(V\alpha^{\mathcal{N}/\mathcal{V}})}(F\alpha^{\mathcal{N}/\mathcal{V}}) \\
([F{\leftarrow}l]\ A_1\ A_2)\alpha^{\mathcal{N}/\mathcal{V}} &= ([F\alpha^{\mathcal{N}/\mathcal{V}}{\leftarrow}l])\ (A_1\alpha^{\mathcal{N}/\mathcal{V}})\ (A_2\alpha^{\mathcal{N}/\mathcal{V}})
\end{aligned}
$$

with

$$
V\alpha^{\mathcal{N}} = \begin{cases} c_i, & \mathit{if}\ V = a_i\ \mathit{and}\ a_i\alpha^{\mathcal{N}} = c_i \\ V, & \mathit{otherwise} \end{cases}
$$

$$
V\alpha^{\mathcal{V}} = \begin{cases} Y_l, & \mathit{if}\ V = X_l\ \mathit{and}\ X\alpha^{\mathcal{V}} = Y \\ V, & \mathit{otherwise} \end{cases}
$$

$$
F\alpha^{\mathcal{N}} = \begin{cases} (G\alpha^{\mathcal{N}})\langle l{=}V\alpha^{\mathcal{N}}\rangle, & \mathit{if}\ F = G\langle l{=}V\rangle \\ (G\alpha^{\mathcal{N}})\cdot(H\alpha^{\mathcal{N}}), & \mathit{if}\ F = G\cdot H \\ (G\alpha^{\mathcal{N}})\backslash(H\alpha^{\mathcal{N}}), & \mathit{if}\ F = G\backslash H \\ F, & \mathit{otherwise} \end{cases}
$$

$$
F\alpha^{\mathcal{V}} = \begin{cases} Y, & \mathit{if}\ X\alpha^{\mathcal{V}} = Y \\ (G\alpha^{\mathcal{V}})\langle l{=}V\alpha^{\mathcal{V}}\rangle, & \mathit{if}\ F = G\langle l{=}V\rangle \\ (G\alpha^{\mathcal{V}})\cdot(H\alpha^{\mathcal{V}}), & \mathit{if}\ F = G\cdot H \\ (G\alpha^{\mathcal{V}})\backslash(H\alpha^{\mathcal{V}}) & \mathit{if}\ F = G\backslash H \\ F, & \mathit{otherwise} \end{cases}
$$

For the rest of this work, the usual convention of writing $a(X)$ is adopted when $a(X).\mathbf{0}$ is meant. An agent $\overline{a}(\langle\rangle)$ sending an empty form can be written as $\overline{a}$, a form $\langle\rangle\langle l = x\rangle$ is written as $\langle l = x\rangle$, $(\nu\ a, b)A$ is an abbreviation for $(\nu\ a)(\nu\ b)A$, and $(\nu\ a_1)...(\nu\ a_n)A$ denotes $(\nu\ \tilde{a})A$. Finally, we will use an underscore '_' as the argument of an input prefix if the argument is not used in the following agent.

7.3.7 Operational semantics

The operational semantics of a process algebra like the FORM calculus is traditionally given in terms of a *labelled transition system* describing the possible evolution of a process. This contrasts with the semantic definition in *term rewriting systems* where an *unlabelled reduction system* is used. The best known term rewriting system is probably the λ-calculus. In the λ-calculus, the reduction of two interacting subterms is only possible if they are in a contiguous position. In process calculi, however, interaction does not depend on a physical contiguity. In other words, in the λ-calculus a *redex* denotes a subterm of a λ-term while a "redex" in a process calculus is usually distributed over the term.

Milner has proposed a guideline for the definition of a *reduction system* for process algebras [Mil90, Mil91]. Using the reduction system technique, axioms for a structural congruence relation are introduced prior the definition of the reduction relation. Basically, this allows us to separate the laws which govern the neighbourhood relation among

1) $A \mid B \equiv B \mid A,\ (A \mid B) \mid C \equiv A \mid (B \mid C),\ A \mid \mathbf{0} \equiv A;$

2) $(\nu\ a)\mathbf{0} \equiv \mathbf{0},\ (\nu\ a)(\nu\ b)A \equiv (\nu\ b)(\nu\ a)A;$

3) $(\nu\ a)A \mid B \equiv (\nu\ a)(A \mid B)$, if a not free in B;

4) $!V(X).A \equiv V(X).A \mid !V(X).A;$

5) $\mathcal{E}(X).A \equiv \mathbf{0},\ \overline{\mathcal{E}}(F) \equiv \mathbf{0};$

Table 7.2: Structural congruence rules for the FORM calculus.

processes for the rules that specify their interaction. Furthermore, this simplifies the presentation of the reduction relation by reducing the number of cases that have to be considered.

The reduction semantics defines the basic mechanisms of computation in a process calculus. The interpretation of the operators is precisely described using the reduction semantics. The reduction relation, however, covers only a part of the behaviour of processes; it describes the behaviour of processes relative to a context in which they are contained. In other words, the reduction semantics describes how two process may interact with each other, but not how these processes (or parts of them) may interact with the environment. Therefore, the reduction relation defines the *interaction* of processes (i.e. their local evolution).

A labelled transition system, on the other hand, describes the possible *intraactions* of processes with the environment. With labelled transition semantics, every possible communication of a process can be determined in a direct way. This allows us to get a simple characterizations of behavioural equivalences. Moreover, with labelled transition semantics, proofs benefit from reasoning in a purely structural way.

Due to the fact that the FORM calculus is based in the $\pi\mathcal{L}$-calculus, the operational semantics and the corresponding rules are very similar. In the following, we will therefore only discuss the aspects which either differ in contrast to the $\pi\mathcal{L}$-calculus or have to be extented in order to cover the extensions made by the FORM calculus. For a detailed discussion of the missing aspects, refer to [Lum99].

Structural congruence. The structural congruence relation '$\equiv$' is the smallest congruence relation over agents that satisfies the axioms given in Table 7.2 and is very similar to the structural congruence rules for the $\pi\mathcal{L}$-calculus.

The axioms 1) to 4) are standard and are the same as for the π-calculus. The axiom 5) defines the behaviour if an empty binding appears in subject position of the leftmost prefix of an agent, implying that such an agent is identical with the inactive agent. This means that a system containing such an agent may reach a deadlock. In general, if the name $\mathcal{E}$ occurs as a subject in the leftmost prefix of an agent, this may be interpreted as a run-time error. However, this is too restrictive in the sense that this view excludes programs which may be useful in some particular contexts.

In order to define a reduction system for the FORM calculus, we have to formally define an evaluation of label matching into a boolean domain. In the following, assume that the form F is normalized. We use $[\![[F \leftarrow l]]\!]$ to denote the evaluation of $[F \leftarrow l]$ into the ordinary two-valued boolean domain $\{\mathit{True},\ \mathit{False}\}$, inductively defined as follows:

$$\begin{array}{rcll} [\![[\langle\rangle \leftarrow l]]\!] & = & \mathit{False} & \\ [\![[F\langle l = a\rangle \leftarrow l]]\!] & = & \mathit{True} & \\ [\![[F\langle m = b\rangle \leftarrow l]]\!] & = & [\![[F \leftarrow l]]\!] & \quad \mathit{if}\ m \neq l \end{array}$$

The reader should note that due to the fact that the form F is normalized, it does not contain any extensions with an empty binding.

The reduction rules for the FORM calculus, which define the interaction of FORM calculus terms, are presented in Table 7.3. The first two rules state that we can reduce under both parallel composition and restriction. The symmetric rule for parallel composition is redundant, because of the use of structural congruence. The communication rule COM takes two agents which are willing to communicate on the channel a and substitutes all form variables X with form F in A. Communication is only allowed for *closed* forms (see the side condition $\mathcal{V}(F) = \emptyset$). The communication rule is the only rule which directly reduces a FORM calculus term. Furthermore, a reduction is not allowed underneath an input prefix since this is the construct that allows sequentialization. The last two rules indicate how to reduce terms in the presence of label matching.

Any communication assumes that all agents involved are in a particular format. The structural congruence rules given in Table 7.2 allow us to rewrite agents so that they have the correct format for applying the corresponding communication rules.

Transition semantics: In the previous paragraph, the semantics of the FORM calculus has been defined using a reduction relation, specifying the actual communication behaviour of agents, and a structural congruence relation. A reduction relation defines how agents my interact with each other; it defines the interaction. The *intraaction*, however, is not covered by the reduction relation. In order to define how agents may interact with the environment, a *labelled transition system* is used that describes the possible interactions with other systems.

As in CCS [Mil89] and the π-calculus [MPW92], a transition in the FORM calculus is of the form

$$A \xrightarrow{\mu} A'$$

Intuitively, this transition means that A can evolve into A' by performing the *action* μ. In the following, μ range over actions that have the following structure:

$$\text{PAR}: \frac{A \longrightarrow A'}{A \mid B \longrightarrow A' \mid B} \qquad \text{RES}: \frac{A \longrightarrow A'}{(\nu\ a)A \longrightarrow (\nu\ a)A'}$$

$$\text{STRUCT}: \frac{A \equiv A' \quad A' \longrightarrow B' \quad B' \equiv B}{A \longrightarrow B}$$

$$\text{COM}: a(X).A \mid \overline{a}.(F) \longrightarrow A\{F/X\}, \quad \textit{if } \mathcal{V}(F) = \emptyset$$

$$\text{TRUE}: \frac{[\![[F\leftarrow l]]\!] = \mathit{True} \quad [F\leftarrow l]\ A\ B}{A} \qquad \text{FALSE}: \frac{[\![[F\leftarrow l]]\!] = \mathit{False} \quad [F\leftarrow l]\ A\ B}{B}$$

Table 7.3: Reduction system for the FORM calculus.

$$\begin{array}{rcll} \mu & = & \tau & \text{silent action} \\ & | & a(\langle \widetilde{l=b} \rangle) & \text{input action} \\ & | & \overline{a}(\langle \widetilde{l=b} \rangle) & \text{output action} \\ & | & (\nu\ \tilde{c})\overline{a}(\langle \widetilde{l=b} \rangle) & \text{restricted output action} \end{array}$$

In the case of input and output, a is the *subject* part, whereas $\langle \widetilde{l=b} \rangle$ is the *object* part of the action. Input and output describe interactions between an agent A and its environment, while the silent action τ is used as placeholder for an internal action in which one subagent of A communicates with another; an external observer can see that something is happening (time is passing), but nothing more.

As mentioned before, only *closed* forms are allowed to be the object of an output action. Furthermore, we only consider *normalized forms* because an external observer can always replace a form with an equivalent, normalized form without changing the behaviour of an agent (refer to Definition 7.4).

The prefix $(\nu\ \tilde{c})$ in a restricted output action is used to record those names in $\langle \widetilde{l=b} \rangle$ that have been created fresh in A (i.e. $\tilde{c} \cap \mathrm{n}(A) = \emptyset$) and are not yet known to the environment. We always assume that $\tilde{c} \subseteq \mathcal{N}(\langle \widetilde{l=b} \rangle)$ and $A \xrightarrow{(\nu\ \tilde{c})\overline{a}(\langle \widetilde{l=b} \rangle)} A'$ means that A emits private names (i.e. names bound in A) along channel a. If some of the names in $\tilde{c}$ are communicated outside of the scope of the ν that binds them, the corresponding ν must be moved outwards in order to include both the sender and the receiver (this is also known as scope extrusion). In order to avoid that bound names are captured in the receiver, this operation may require an α-conversion of bound names in the receiver.

In the following, the silent action, the input action, and the output action will be called *free* actions, while the restricted output actions will be called *bound* actions (or simply bounded output). Given an action μ, the bound and free names of μ, written $\mathrm{bn}(\mu)$ and $\mathrm{fn}(\mu)$ are defined below. The names of μ, written $\mathrm{n}(\mu)$, are $\mathrm{bn}(\mu) \cup \mathrm{fn}(\mu)$.

$$\text{IN} : a(X).A \xrightarrow{a(\langle \widetilde{l=b} \rangle)} A\{\langle \widetilde{l=b} \rangle / X\} \qquad \text{OUT} : \overline{a}(\langle \widetilde{l=b} \rangle) \xrightarrow{\overline{a}(\langle \widetilde{l=b} \rangle)} \mathbf{0}$$

$$\text{OPEN} : \frac{A \xrightarrow{(\nu\ \tilde{c})\overline{a}(\langle \widetilde{l=b} \rangle)} A' \qquad d \neq a \qquad d \in \mathcal{N}(\langle \widetilde{l=b} \rangle) - \tilde{c}}{(\nu\ d)A \xrightarrow{(\nu\ d, \tilde{c})\overline{a}(\langle \widetilde{l=b} \rangle)} A'}$$

$$\text{CLOSE} : \frac{A \xrightarrow{(\nu\ \tilde{c})\overline{a}(\langle \widetilde{l=b} \rangle)} A' \qquad B \xrightarrow{a(\langle \widetilde{l=b} \rangle)} B' \qquad \tilde{c} \notin \text{fn}(B)}{A \mid B \xrightarrow{\tau} (\nu\ \tilde{c})(A' \mid B')}$$

$$\text{COM} : \frac{A \xrightarrow{\overline{a}(\langle \widetilde{l=b} \rangle)} A' \qquad B \xrightarrow{a(\langle \widetilde{l=b} \rangle)} B'}{A \mid B \xrightarrow{\tau} A' \mid B'}$$

$$\text{REPL} : \frac{a(X).A \xrightarrow{a(\langle \widetilde{l=b} \rangle)} A\{\langle \widetilde{l=b} \rangle / X\}}{!a(X).A \xrightarrow{a(\langle \widetilde{l=b} \rangle)} A\{\langle \widetilde{l=b} \rangle / X\} \mid !a(X).A}$$

$$\text{PAR} : \frac{A \xrightarrow{\mu} A' \qquad \text{bn}(\mu) \cap \text{fn}(B) = \emptyset}{A \mid B \xrightarrow{\mu} A' \mid B} \qquad \text{RES} : \frac{A \xrightarrow{\mu} A' \qquad a \notin \text{n}(\mu)}{(\nu\ a)A \xrightarrow{\mu} (\nu\ a)A'}$$

$$\text{TRUE} : \frac{[\![[F \leftarrow l]]\!] = \mathit{True} \quad A \xrightarrow{\mu} A'}{[F \leftarrow l]\ A\ B \xrightarrow{\mu} A'} \qquad \text{FALSE} : \frac{[\![[F \leftarrow l]]\!] = \mathit{False} \quad B \xrightarrow{\mu} B'}{[F \leftarrow l]\ A\ B \xrightarrow{\mu} B'}$$

Table 7.4: Labelled transition system for the FORM calculus.

$$\begin{array}{ll}
\text{bn}(a(\langle \widetilde{l=b} \rangle)) \ = \ \emptyset & \text{fn}(a(\langle \widetilde{l=b} \rangle)) \ = \ \{a\} \cup \mathcal{N}(\langle \widetilde{l=b} \rangle) \\
\text{bn}(\overline{a}(\langle \widetilde{l=b} \rangle)) \ = \ \emptyset & \text{fn}(\overline{a}(\langle \widetilde{l=b} \rangle)) \ = \{a\} \cup \mathcal{N}(\langle \widetilde{l=b} \rangle) \\
\text{bn}((\nu\ \tilde{c})\overline{a}(\langle \widetilde{l=b} \rangle)) \ = \ \{\tilde{c}\} & \text{fn}((\nu\ \tilde{c})\overline{a}(\langle \widetilde{l=b} \rangle)) \ = \ (\{a\} \cup \mathcal{N}(\langle \widetilde{l=b} \rangle)) - \{\tilde{c}\} \\
\text{bn}(\tau) \ = \ \emptyset & \text{fn}(\tau) \ = \ \emptyset
\end{array}$$

The FORM calculus early transition system is presented in Table 7.4. When an action from agent $a(X).A$ is inferred, the variable X is instantiated at the time of inferring the input transition (rule IN). This allows us to define a bisimulation for the FORM calculus without clauses for name-instantiation (similar to the bisimulation for the $\pi\mathcal{L}$-calculus [Lum99]). Instantiation should be considered as the mechanism that first substitutes X and then applies all projections X_l for some label l. The symmetric versions of rules PAR and COM have been omitted. In fact, parallel composition should be understood as a commutative operator.

7.3.8 Observable equivalence

An important question in the theory of process calculi is when two processes can be said to exhibit the same behaviour. As in the λ-calculus, the most intuitive way of defining an equivalence of processes is via some notion of *contextual equivalence*. A *process context* $\mathcal{C}[\cdot]$ is a process expression with a hole into which one can place another process. We say that the processes A and B are equivalent when $\mathcal{C}[A]$ and $\mathcal{C}[B]$ have the same *observable behaviour* for each process context $\mathcal{C}[\cdot]$.

However, the definition of a contextual equivalence between two processes can be difficult to establish, but there is an alternative to contextual equivalence which is based on direct conditions on the processes themselves. Given a *labelled transition semantics*, there is a standard definition of bisimulation equivalence which can be applied to such a transition system [Mil89]. Moreover, bisimulation equivalence is widely considered to be the finest equivalence needed to study transition systems.

Basically, bisimulation defines equivalence as mutual simulation of transitions of processes resulting in equivalent states. Formally, a binary relation $\mathcal{R}$ is a (ground) bisimulation on processes such that for arbitrary action μ, $A \mathcal{R} B$ implies

- whenever $A \xrightarrow{\mu} A'$, then B' exists such that $B \xrightarrow{\mu} B'$ and $A' \mathcal{R} B'$,
- whenever $B \xrightarrow{\mu} B'$, then A' exists such that $A \xrightarrow{\mu} A'$ and $A' \mathcal{R} B'$.

The main idea of a bisimulation is that an external observer performs experiments with two processes A and B, observing the results in turn in order to match the two processes' behaviours step-by-step. Furthermore, the definition of bisimulation is given in a coinductive style (i.e. two processes are considered to be bisimular if it is not possible to show that they are not).

Many variants of bisimulation have been proposed (e.g. early, late, open, and barbed bisimulation [MPW92, MS92, San93, San96a]). All variants, however, distinguish between a *strong* and a *weak* definition of bisimulation. The difference is that in the weak case, an arbitrary number of silent transitions are regarded as equivalent to a single transition. Therefore, the weak bisimulation is strictly coarser than the strong bisimulation in the sense that whenever two processes A and B are strongly bisimular, they are also weakly bisimular. In practice, weak bisimulation is often more useful, since we typically want to regard two processes to be equivalent if they have the same *observable* behaviour even if one performs more silent transitions than the other.

In a calculus with synchronous output, the existence of an input transition precisely models the success of an observer's input-experiment. In the synchronous case, input actions for a process A are only generated if there exists a matching receiver that is enabled inside A. The existence of an input transition such that A evolves to A' reflects precisely the fact that a message offered by the observer has actually been consumed.

In a calculus with asynchronous output, however, the situation is slightly different. The sender of an output message does not know when the message is actually consumed due to the fact that at the time of the consumption of the message, its sender is not participating in the event anymore. Therefore, an asynchronous observer, in contrast to a synchronous one, cannot directly detect the input actions of the observed process. Consequently, a calculus with asynchronous output requires an appropriate semantic framework based on an asynchronous experimenter and, therefore, a different notion of input-experiment is needed.

Amadio et al. [ACS96] proposed a solution for asynchronous observation based on the standard labelled transition system. In their approach, the asynchronous style of input-experiments is directly incorporated into the definition of bisimulation, and inputs of processes have to be simulated indirectly by observing the output behaviour of a process in the context of arbitrary messages. This is the approach we follow for the FORM calculus.

In order to define bisimulation on the approach discussed above, it is necessary to precisely define the notions of strong and week transitions, respectively. For the rest of this work, weak arrows $\Longrightarrow$ denote the reflexive and transitive closure of transitions:

$$\begin{array}{ll} A \overset{\tau}{\Longrightarrow} A' & \text{iff } A(\overset{\tau}{\longrightarrow}) * A' \\ A \overset{\mu}{\Longrightarrow} A' & \text{iff } A \overset{\tau}{\Longrightarrow} \cdot \overset{\mu}{\longrightarrow} \cdot \overset{\tau}{\Longrightarrow} A',\ \mu \neq \tau \end{array}$$

Definition 7.14 ($\mathcal{F}$-bisimulation) *A binary relation $\mathcal{R}$ over closed agents A and B is a strong $\mathcal{F}$-bisimulation if it is symmetric and $A\ \mathcal{R}\ B$ implies:*

- *whenever $A \overset{\mu}{\longrightarrow} A'$, where μ is either τ or output with $\mathrm{bn}(\mu) \cap \mathrm{fn}(A|B) = \emptyset$, then B' exists such that $B \overset{\mu}{\longrightarrow} B'$ and $A'\ \mathcal{R}\ B'$,*
- *$(A \mid \overline{a}(\langle \widetilde{l = b} \rangle))\ \mathcal{R}\ (B \mid \overline{a}(\langle \widetilde{l = b} \rangle))$ for all messages $\overline{a}(\langle \widetilde{l = b} \rangle)$.*

Two agents A and B are strongly bisimular, written $A \overset{\mathcal{F}}{\sim} B$, if they are related by a strong bisimulation. The notion of weak $\mathcal{F}$-bisimulation, written $\overset{\mathcal{F}}{\approx}$, is obtained by replacing strong transitions with weak transitions. Two agents A and B are weakly bisimular, written $A \overset{\mathcal{F}}{\approx} B$, if there exists a weak $\mathcal{F}$-bisimulation $\mathcal{R}$ with $A\ \mathcal{R}\ B$.

We call $\mathcal{R}$ as above a $\mathcal{F}$-bisimulation. Then both $\overset{\mathcal{F}}{\sim}$ and $\overset{\mathcal{F}}{\approx}$ are the union of all strong and weak $\mathcal{F}$-bisimulations, respectively. Furthermore, both $\overset{\mathcal{F}}{\sim}$ and $\overset{\mathcal{F}}{\approx}$ require preservation under parallel composition with an output. For the rest of this work, we are mainly interested in comparing systems by considering only their *observable* behaviour. Therefore, we abstract from silent actions and use the observation equivalence $\overset{\mathcal{F}}{\approx}$ as the main equivalence for the FORM calculus.

As in the $\pi\mathcal{L}$-calculus, the lack of summation allows us to establish congruence for $\overset{\mathcal{F}}{\approx}$. Furthermore, unlike in the π-calculus, names are are always constant (i.e. we do not have name substitution). Therefore, in an input-prefixed agent $a(X).A$, only the form variable X is substituted by a received form $\langle \widetilde{l = b} \rangle$, and this substitution does not change any name in A. As a consequence, if A is a closed agent (i.e. $\mathrm{fv}(A) = \emptyset$), it is possible to add an arbitrary number of input prefixes without changing the behaviour of A.

In order to show that $\overset{\mathcal{F}}{\approx}$ is a congruence relation, we have to show that it is an equivalence relation and that bisimulation is preserved under all the operators of the calculus.

Due to the similarity of the FORM calculus and the $\pi\mathcal{L}$-calculus, most of the necessary proofs are similar and have been omitted here (they are included in appendix B). The only new proof which needs to be added is the preservation of bisimulation under matching, which is the topic of the following proposition.

Proposition 7.6 *For any closed agents A, B, and C:*

$$\begin{array}{lcl} A \overset{\mathcal{F}}{\approx} C \ \wedge\ [\![[F\!\leftarrow\! l]]\!] = \mathit{True} & \Rightarrow & [F\!\leftarrow\! l]\ A\ B \overset{\mathcal{F}}{\approx} [F\!\leftarrow\! l]\ C\ B, \quad \mathit{and} \\ B \overset{\mathcal{F}}{\approx} C \ \wedge\ [\![[F\!\leftarrow\! l]]\!] = \mathit{False} & \Rightarrow & [F\!\leftarrow\! l]\ A\ B \overset{\mathcal{F}}{\approx} [F\!\leftarrow\! l]\ A\ C. \end{array}$$

PROOF: We show that the two relations

$$\begin{array}{lcl} \mathcal{R}_1 & = & \{ ([F\!\leftarrow\! l]\ A\ B\ ,\ [F\!\leftarrow\! l]\ A\ C) \mid [\![[F\!\leftarrow\! l]]\!] = \mathit{True} \ \wedge\ A \overset{\mathcal{F}}{\approx} C \} \cup \overset{\mathcal{F}}{\approx} \\ \mathcal{R}_2 & = & \{ ([F\!\leftarrow\! l]\ A\ B\ ,\ [F\!\leftarrow\! l]\ A\ C) \mid [\![[F\!\leftarrow\! l]]\!] = \mathit{False} \ \wedge\ B \overset{\mathcal{F}}{\approx} C \} \cup \overset{\mathcal{F}}{\approx} \end{array}$$

are both weak $\mathcal{F}$-bisimulations.

For $\mathcal{R}_1$, consider τ or output actions with $\mathrm{bn}(\mu) \cap \mathrm{fn}(A|B) = \emptyset$:

> $[F\!\leftarrow\! l]\ A\ B \overset{\mu}{\Longrightarrow} A'$ is inferred from $A \overset{\mu}{\Longrightarrow} A'$ and $[\![[F\!\leftarrow\! l]]\!] = \mathit{True}$. Since $A \overset{\mathcal{F}}{\approx} C$, this implies $C \overset{\mu}{\Longrightarrow} C'$ with $A' \overset{\mathcal{F}}{\approx} C'$. Then $[F\!\leftarrow\! l]\ C\ B \overset{\mu}{\Longrightarrow} C'$ is the required move, since $(A'\ ,\ C') \in \mathcal{R}_1$.

For $\mathcal{R}_1$, consider input actions:

> $[F\!\leftarrow\! l]\ A\ B \overset{a(\langle \widetilde{l\!=\!b} \rangle)}{\Longrightarrow} A'$ is inferred from $A \overset{a(\langle \widetilde{l\!=\!b} \rangle)}{\Longrightarrow} A'$ and $[\![[F\!\leftarrow\! l]]\!] = \mathit{True}$. Since $A \overset{\mathcal{F}}{\approx} C$ and by definition $(A|\overline{a}(\langle \widetilde{l\!=\!b} \rangle), B|\overline{a}(\langle \widetilde{l\!=\!b} \rangle)) \in \mathcal{R}_1$ for all messages $\overline{a}(\langle \widetilde{l\!=\!b} \rangle)$, this implies $C \overset{a(\langle \widetilde{l\!=\!b} \rangle)}{\Longrightarrow} C'$ with $A' \overset{\mathcal{F}}{\approx} C'$. Then $[F\!\leftarrow\! l]\ C\ B \overset{a(\langle \widetilde{l\!=\!b} \rangle)}{\Longrightarrow} C'$ is the required move, since $(A'\ ,\ C') \in \mathcal{R}_1$.

The proof for $\mathcal{R}_2$ is similar. $\square$

7.3.9 α-conversion

In this section we show that alpha-convertible agents are weakly $\mathcal{F}$-bisimular. In order to prove that $\equiv_\alpha$ is a weak $\mathcal{F}$-bisimulation, we use the Lemma given below.

Lemma 7.1 *Suppose that $A \equiv_\alpha B$.*

- *If μ is not a restricted output action and $A \overset{\mu}{\Longrightarrow} A'$, then there exists a B' with $A' \equiv_\alpha B'$ and $B \overset{\mu}{\Longrightarrow} B'$.*
- *If $A \overset{(\nu\ \tilde{c})\overline{a}(\langle \widetilde{l\!=\!b} \rangle)}{\Longrightarrow} A'$ and $\tilde{d} \cap \mathrm{n}(B) = \emptyset$, then there exists a B' with $A'\{\tilde{d}/\tilde{c}\}^{\mathcal{N}}_{\alpha} \equiv_\alpha B'$ and $B \overset{(\nu\ \tilde{d})\overline{a}(\langle \widetilde{l\!=\!b} \rangle)}{\Longrightarrow} B'$.*

PROOF: The proof is by induction on the depth of inference. We consider each transition rule as the last rule applied in the inference of $A \overset{\mu}{\Longrightarrow} A'$.

- IN: We have $\mu = a(\widetilde{\langle l = b \rangle})$, $\mathrm{bn}(\mu) = \emptyset$, $A \equiv a(X).A_1$, and $A' \equiv A_1\{\widetilde{\langle l = b \rangle}/X\}$. Since $A \equiv_\alpha B$, B must also be an input-prefixed agent, differing at most in the bound variable of the input prefix. By applying alpha-conversion, it is possible to make the prefixes identical (i.e. $B \equiv a(X).B_1$). It immediately follows that B has a transition $a(\widetilde{\langle l = b \rangle})$ with $B \overset{a(\widetilde{\langle l = b \rangle})}{\Longrightarrow} B'$, $B' \equiv B_1\{\widetilde{\langle l = b \rangle}/X\}$ and $A' \equiv_\alpha B'$.
- OUT: We have $\mu = \overline{a}(\widetilde{\langle l = b \rangle})$, $\mathrm{bn}(\mu) = \emptyset$, $A \equiv \overline{a}(\widetilde{\langle l = b \rangle})$, and $A' \equiv \mathbf{0}$. Since $\mathrm{bn}(A) = \emptyset$ and $\mathrm{bv}(A) = \emptyset$, it is possible to replace $A \equiv_\alpha B$ with $A \equiv B$. It also holds that $B \overset{\overline{a}(\widetilde{\langle l = b \rangle})}{\Longrightarrow} B'$ with $A' \equiv_\alpha B'$.
- OPEN: We have $\mu = (\nu\, b, \tilde{c})\overline{a}(\widetilde{\langle l = b \rangle})$, $\mathrm{bn}(\mu) = \{b\} \cup \tilde{c}$, $A \equiv (\nu\, b)A_1$, and $A' \equiv A_1'$. By assumption we can prove that $A_1 \overset{(\nu\ \tilde{c})\overline{a}(\widetilde{\langle l = b \rangle})}{\Longrightarrow} A'$, and for some fresh name b we have $b \neq a$ and $b \in \mathcal{N}(\widetilde{\langle l = b \rangle}) - \tilde{c}$. Furthermore, it holds that $\mathrm{fn}(A) = \mathrm{fn}(A_1)$ and since $A \equiv_\alpha B$, it also holds $\mathrm{fn}(A) = \mathrm{fn}(B) = \mathrm{fn}(B_1)$ with $B_1 \equiv_\alpha A_1\{d/b\}^{\mathcal{N}}_\alpha$. Since $A_1 \overset{(\nu\ \tilde{c})\overline{a}(\widetilde{\langle l = b \rangle})}{\Longrightarrow} A'$, we can prove for some new name d with $d \neq a$ and $d \in \mathcal{N}(\widetilde{\langle l = b \rangle}\{d/b\}^{\mathcal{N}}_\alpha) - \tilde{c}$ that $B_1 \overset{(\nu\ \tilde{c})\overline{a}(\widetilde{\langle l = b \rangle})}{\Longrightarrow} B'$ with $B' \equiv (\nu\ d)B_1 \equiv_\alpha A'\{d/b\}^{\mathcal{N}}_\alpha$. Therefore, it also holds that $(\nu\ d)B_1 \overset{(\nu\ d, \tilde{c})\overline{a}(\widetilde{\langle l = b \rangle})}{\Longrightarrow} B'$.
- COM: We have $\mu = \tau$, $A \equiv A_1|A_2$ with $A_1 \overset{\overline{a}(\widetilde{\langle l = b \rangle})}{\Longrightarrow} A_1'$, $A_2 \overset{a(\widetilde{\langle l = b \rangle})}{\Longrightarrow} A_2'$, and $A' \equiv A_1'|A_2'$. By assumption we can prove that $A_1 \overset{\overline{a}(\widetilde{\langle l = b \rangle})}{\Longrightarrow} A_1'$ and $A_2 \overset{a(\widetilde{\langle l = b \rangle})}{\Longrightarrow} A_2'$. Since $A \equiv_\alpha B$, it holds that $\mathrm{fn}(A) = \mathrm{fn}(B)$. With $B_1 \equiv_\alpha A_1$, $B_2 \equiv_\alpha A_2$, $B_1' \equiv_\alpha A_1'$, and $B_2' \equiv_\alpha A_2'$, it is possible to prove that $B_1 \overset{\overline{a}(\widetilde{\langle l = b \rangle})}{\Longrightarrow} B_1'$ and $B_2 \overset{a(\widetilde{\langle l = b \rangle})}{\Longrightarrow} B_2'$. Therefore, it also holds that $B \overset{\tau}{\Longrightarrow} B'$ with $B \equiv B_1|B_2$ and $B' \equiv B_1'|B_2'$.
- TRUE: We have $A \equiv [F \leftarrow l]\ A_1\ A_2$ and $[\![[F \leftarrow l]]\!] = \mathit{True}$. By assumption we can prove that $A_1 \overset{\mu}{\Longrightarrow} A_1'$. Since $A \equiv_\alpha B$, we have $B \equiv [F \leftarrow l]\ B_1\ B_2$ with $A_1 \equiv_\alpha B_1$. Now we can prove that $B_1 \overset{\mu}{\Longrightarrow} B_2'$, implying that $B \equiv [F \leftarrow l]\ B_1\ B_2 \overset{\mu}{\Longrightarrow} B_1'$ also holds. Note that $A_2 \equiv_\alpha B_2$ is not needed.

The remaining proofs for FALSE, CLOSE, PAR, RES, and REPL are similar (refer to [MPW92] for proof sketches). □

If A is a closed agent and $B \equiv_\alpha A$, then B is also closed. Therefore, A and B differ at most in the choice of bound names and variables, respectively. By applying alpha-conversion, it is possible to make them identical. Using Lemma 7.1, every move of A can be matched up by $B \equiv_\alpha A\alpha^{\mathcal{N}/\mathcal{V}}$ for some alpha-substitution $\alpha^{\mathcal{N}/\mathcal{V}}$. Therefore, we have shown that $\equiv_\alpha$ is a weak $\mathcal{F}$-bisimulation.

The reader should note that from the fact that $\equiv_\alpha$ is a weak $\mathcal{F}$-bisimulation, it immediately follows that alpha-conversion is only defined on closed agents.

7.3.10 Encoding the mini π-calculus in the FORM calculus

In the following two sections, we present an encoding of the mini π-calculus introduced in section 7.1 in the FORM calculus and vice versa. The encodings illustrate that both calculi can faithfully encode each other. Moreover, we show that both encodings preserve the weak asynchronous bisimulation relation (i.e. if two asynchronous π-processes are weakly bisimular, then their encodings are also weakly bisimular in the FORM calculus and vice versa).

The encoding of the mini π-calculus in the FORM calculus follows a similar scheme than the corresponding translation into the $\pi\mathcal{L}$-calculus [Lum99]. The basic idea of our encoding is to use de Bruijn indices [dB72]. More precisely, in an input-prefixed process $a(\tilde{x}).P$, we assign every parameter name x_i a unique non-negative integer i with respect to a fresh form variable X (in fact, the parameter's position index is used) and replace every application of x_i in P by a projection X_{l_i} where l_i maps to x_i.

Similarly, in an output-particle $\overline{a}\langle\tilde{b}\rangle$, we replace every b_i with a binding $\langle l_i = b_i \rangle$ where i is a unique non-negative integer (i denotes the actual output parameter position). The reader should note that if b_i is bound by an input prefix $a(\tilde{x})$, then the encoding replaces b_i by X_{l_j}. In X_{l_j}, j is the position index of b_i in the input prefix (i.e. $b_i = x_j$) and l_j denotes b_i with respect to the fresh form variable X. For example, the process

$$a(x_1, x_2, x_3).(\overline{b}\langle x_1, x_2\rangle \mid \overline{c}\langle x_1, x_3\rangle)$$

is encoded as the agent

$$a(X).(\overline{b}(\langle l_1 = X_{l_1}\rangle\langle l_2 = X_{l_2}\rangle) \mid \overline{c}(\langle l_1 = X_{l_1}\rangle\langle l_2 = X_{l_3}\rangle))$$

The encoding uses the function $\mathcal{C}[\![\ ]\!]^{\pi\mathcal{F}}_{\Lambda,\Gamma}$ which maps a mini π-process P to a corresponding agent in the FORM calculus. Within the encoding, Γ is used to record all names used in mini π-processes. Γ can be considered as a symbol table that maps restricted names to themselves and names that are bound by an input prefix to a corresponding projection. In the function $\mathcal{C}[\![\ ]\!]^{\pi\mathcal{F}}_{\Lambda,\Gamma}$, Λ is an input counter (i.e. Λ keeps track of all input-prefixes) and denotes the actual fresh form variable that replaces the input parameters $\tilde{x}$ of an input-prefixed process $a(\tilde{x}).P$. As an example, consider the encoding of the process

$$a(x_1).b(y_1).\overline{c}\langle x_1, y_1\rangle.$$

As a starting point, $\Lambda = 0$ is used such that first the input prefix becomes $a(X^0)$. Since Λ is equal to 1 for the second input prefix, the encoding yields $b(X^1)$. Therefore, the encoded agent becomes

$$a(X^0).b(X^1).\overline{c}(\langle l_1 = X^0_{l_1}\rangle\langle l_1 = X^1_{l_1}\rangle)$$

The encoding function $\mathcal{C}[\![\ ]\!]^{\pi\mathcal{F}}_{\Lambda,\Gamma}$ is presented in Table 7.5. It starts starts with $\Lambda = 0$ and $\Gamma = \emptyset$. The encoding function for mini π-calculus actions is denoted with $\mathcal{C}[\![\alpha]\!]^{\pi\mathcal{F}}$. Furthermore, $\mathcal{C}[\![\ ]\!]^{\pi\mathcal{F}}_{\Lambda,\Gamma}$ uses a function $\mathbf{map}$ that maps names to names and projections, respectively.

$$
\begin{array}{rcl}
\mathcal{C}[\![\mathbf{0}]\!]^{\pi\mathcal{F}}_{\Lambda,\Gamma} & = & \mathbf{0} \\
\mathcal{C}[\![P_1 \mid P_1]\!]^{\pi\mathcal{F}}_{\Lambda,\Gamma} & = & \mathcal{C}[\![P_1]\!]^{\pi\mathcal{F}}_{\Lambda,\Gamma} \mid \mathcal{C}[\![P_2]\!]^{\pi\mathcal{F}}_{\Lambda,\Gamma} \\
\mathcal{C}[\![(\nu\, x)P]\!]^{\pi\mathcal{F}}_{\Lambda,\Gamma} & = & (\nu\, x)\mathcal{C}[\![P]\!]^{\pi\mathcal{F}}_{\Lambda,\Gamma:(x\mapsto n(x))} \\
\mathcal{C}[\![!a(x_1,\ldots,x_n).P]\!]^{\pi\mathcal{F}}_{\Lambda,\Gamma} & = & !\mathcal{C}[\![a(x_1,\ldots,x_n).P]\!]^{\pi\mathcal{F}}_{\Lambda,\Gamma} \\
\mathcal{C}[\![a(x_1,\ldots,x_n).P]\!]^{\pi\mathcal{F}}_{\Lambda,\Gamma} & = & \mathbf{map}(\Gamma,a)(X^{\Lambda+1}).\mathcal{C}[\![P]\!]^{\pi\mathcal{F}}_{\Lambda+1,\Gamma:(x_1\mapsto v(\Lambda+1,1)):\ldots:(x_n\mapsto v(\Lambda+1,n))} \\
\mathcal{C}[\![\overline{a}\langle x_1,\ldots,x_n\rangle]\!]^{\pi\mathcal{F}}_{\Lambda,\Gamma} & = & \overline{\mathbf{map}(\Gamma,a)}(\langle l_1 = \mathbf{map}(\Gamma,x_1)\rangle\ldots\langle l_n = \mathbf{map}(\Gamma,x_n)\rangle)
\end{array}
$$

$$
\mathbf{map}(\Gamma,x) = \begin{cases} X^{\Lambda}_{l_n}, & \mathit{if}\ \Gamma = \Gamma_1 : (x \mapsto v(\Lambda,n)) : \Gamma_2 \\ x, & \mathit{otherwise} \end{cases}
$$

$$
\begin{array}{rcl}
\mathcal{C}[\![\tau]\!]^{\pi\mathcal{F}} & = & \tau \\
\mathcal{C}[\![a(\tilde{x})]\!]^{\pi\mathcal{F}} & = & a(\langle \widetilde{l=b}\rangle) \\
\mathcal{C}[\![\overline{a}\langle\tilde{x}\rangle]\!]^{\pi\mathcal{F}} & = & \overline{a}(\langle \widetilde{l=b}\rangle) \\
\mathcal{C}[\![(\nu\,\tilde{c})\overline{a}\langle\tilde{b}\rangle]\!]^{\pi\mathcal{F}} & = & (\nu\,\tilde{c})\overline{a}(\langle \widetilde{l=b}\rangle)
\end{array}
$$

Table 7.5: Encoding of the mini π-calculus in the FORM calculus.

Note that the extension of Γ may hide existing mappings. As an example, consider the situation where Γ is extended by $(x \mapsto v(\Lambda,n))$ and Γ already contains a mapping for x ($(x \mapsto n(x))$ such that $\Gamma = \Gamma_1 : (x \mapsto n(x)) : \Gamma_2$). In such a situation, the function $\mathbf{map}(\Gamma : (x \mapsto v(\Lambda,n)), x)$ yields $X^{\Lambda}_{l_n}$, which corresponds to the latest mapping $(x \mapsto v(\Lambda,n))$. Furthermore, if Γ defines no mapping for a name x, then $\mathbf{map}(\Gamma,x) = x$. The collection of names $x_1,\ldots,x_n$ for which a mapping is declared in Γ is indicated by $dom(\Gamma)$.

All properties which hold for the translation of the $\pi\mathcal{L}$-calculus into the π-calculus also hold for the encoding defined in this section. In particular, if two π-processes P_1 and P_2 are weakly 1-bisimular (written $P_1 \approx P_2$) [ACS96], then the encodings $\mathcal{C}[\![P_1]\!]^{\pi\mathcal{F}}_{\Lambda,\Gamma}$ and $\mathcal{C}[\![P_2]\!]^{\pi\mathcal{F}}_{\Lambda,\Gamma}$ are weakly $\mathcal{F}$-bisimular (i.e. $\mathcal{C}[\![P_1]\!]^{\pi\mathcal{F}}_{\Lambda,\Gamma} \overset{\mathcal{F}}{\approx} \mathcal{C}[\![P_2]\!]^{\pi\mathcal{F}}_{\Lambda,\Gamma}$). The corresponding proofs are very similar to the ones presented in [Lum99] and have therefore been omitted here.

7.3.11 Encoding the FORM calculus in the mini π-calculus

In this section, we present an encoding of the FORM calculus in the mini π-calculus. We use the approach to adapt the translation of the $\pi\mathcal{L}$-calculus into the mini π-calculus and extend this translation with encodings for polymorphic restriction as well as matching. We also show that the resulting encoding is faithful.

The basic idea of our encoding is to represent forms and agents of the FORM calculus as processes in the mini π-calculus, similar to the translation of the $\pi\mathcal{L}$-calculus into the mini π-calculus [Lum99]. More precisely, a form is encoded as a mini π-process listening at a channel f (the *location* of the form) for a channel L that represents the actual projection label, a result channel R along which the result of the projection is returned, and an *error-continuation* channel E being the location of the continuation if the actual label is not defined for the form located at f. As an example, the form

$$\langle l = a \rangle \langle m = b \rangle$$

is encoded as the following replicated process:

$$\begin{array}{l} P \equiv\, !f(L,R,E).(\nu\, m,c)(\, \overline{L}\langle c,m\rangle \\ \qquad\qquad\qquad\qquad\quad \mid m(\_).\overline{R}\langle b\rangle \\ \qquad\qquad\qquad\qquad\quad \mid c(\_).(\nu\, f')(\, \overline{f'}\langle L,R,E\rangle \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid f'(L,R,E). \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\nu\, l,c')(\, \overline{L}\langle l,c'\rangle \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid l(\_).\overline{R}\langle a\rangle \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid c'(\_).(\nu\, f'')(\, \overline{f''}\langle L,R,E\rangle \mid f''(L,R,E).\overline{E}\langle\rangle))))). \end{array}$$

f denotes the location of the form $\langle l = a\rangle\langle m = b\rangle$. The output-particles $\overline{L}\langle c, m\rangle$ and $\overline{L}\langle l, c'\rangle$ initiate the test for a label: if L denotes label l, then $l(\_).\overline{R}\langle a\rangle$ is triggered as continuation. Similarly, if L denotes label m, then $m(\_).\overline{R}\langle b\rangle$ is triggered as continuation. This scheme roughly corresponds to Milner's encoding of boolean values [Mil91]. If L denotes neither l nor m, then $\overline{E}\langle\rangle$ is triggered in order to indicate that a run-time error has occurred (i.e. an *undefined label* has been accessed).

The encoding of an extension with an empty binding is similar to the encoding of the form described above. As an example, consider the encoding of the form $\langle l = a\rangle\langle m = \mathcal{E}\rangle$ given below:

$$\begin{array}{l} P \equiv\, !f(L,R,E).(\nu\, m,c)(\, \overline{L}\langle c,m\rangle \\ \qquad\qquad\qquad\qquad\quad \mid m(\_).\overline{E}\langle\rangle \\ \qquad\qquad\qquad\qquad\quad \mid c(\_).(\nu\, f')(\, \overline{f'}\langle L,R,E\rangle \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid f'(L,R,E). \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\nu\, l,c')(\, \overline{L}\langle l,c'\rangle \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid l(\_).\overline{R}\langle a\rangle \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid c'(\_).(\nu\, f'')(\, \overline{f''}\langle L,R,E\rangle \mid f''(L,R,E).\overline{E}\langle\rangle))))). \end{array}$$

f denotes the location of the form $\langle l = a\rangle\langle m = \mathcal{E}\rangle$, where the output-particles $\overline{L}\langle c, m\rangle$ and $\overline{L}\langle l, c'\rangle$ again initiate the test for a label. If L denotes label l, then $l(\_).\overline{R}\langle a\rangle$ is triggered as continuation (like in the example above). However, if L denotes label m (which has been "removed" from the form), then the continuation is triggered on the error channel E, indicating that the form does not define a binding for label m. The encoding of polymorphic restriction is similar (refer to Table 7.7).

Labels are encoded as replicated processes that wait for a tuple of continuation channels and signal along their designated label channel. Without loss of generality, labels are encoded as names. If there is a conflict with an existing name in an agent A to be encoded, A is α-converted using fresh names. For the form $\langle l = a\rangle\langle m = b\rangle$, two label processes have to be defined, where x_1 is used as continuation channel for label l and x_2 as continuation channel for label m, respectively. Hence, the encoding of the form $\langle l{=}a\rangle\langle m{=}b\rangle$ is defined as follows:

$$!l(x_1, x_2).\overline{x_1}\langle\rangle$$
$$!m(x_1, x_2).\overline{x_2}\langle\rangle$$

The encoding of agents is relatively straightforward. For an input-prefixed agent $a(X).A$, the input prefix $a(X)$ is also used as an input prefix for the corresponding π-process. The reader should note that the form variable X is encoded as a name.

An output-particle $\overline{a}(F)$ (for simplicity, a simple name is used here) is encoded as the π-process $(\nu\ f)(\overline{a}\langle f\rangle \mid P)$, where $\overline{a}\langle f\rangle$ emits along the original channel a the location of the form process (located at channel f). The right-hand side process P implements the form F.

A projection Y_l is encoded as a parallel composition of an output-particle $\overline{Y}\langle l, r, \mathit{Error}\rangle$ and an input-prefixed process $r(u_l).P$. The former process triggers the name projection process for Y and sends along a fresh channel r the value of the projection if label l is defined. If no binding for the label l is defined, a corresponding message is sent along the error-continuation channel Error. The input-prefixed process $r(u_l).P$ instantiates a name u_l in P which binds the value received along channel r (i.e. the value of the projection Y_l). The reader should note that $Error$ is the location of the process that handles name projections for labels that are not defined in a form.

The encoding of $[F \leftarrow l]\ A_1\ A_2$ builds upon the projection described above and is encoded as a parallel composition of four processes: the first two processes encode a name projection on the form F of the matching whereas the last two processes correspond to the encoding of the continuation processes for a successful and failed match, respectively. In addition, the encoding creates two fresh channels r and r' which are used to parameterize the name projection. The continuation of the projection is triggered on channel r if the form F defines a binding for a label l, on channel r' otherwise. The encoding of A_1 defines an input prefix on channel r, indicating that A_1 is triggered when F defines a binding for label l. Similarly, the encoding of A_2 has an input prefix on channel r'.

The encoding of the FORM calculus in the mini π-calculus is defined using the function $\mathcal{C}[\![\ ]\!]_{\phi}^{\mathcal{F}\pi}$ which maps an agent of the FORM calculus to a corresponding mini π-process (refer to Table 7.6). In the encoding, we use a function ϕ in order to map the set $\mathcal{L}$ of labels to the set $I\!N$ of natural numbers. Without losing generality, ϕ maps $\mathcal{L}$ to $\{1, 2, \ldots, n\}$, where n is the number of labels used in the agent to be translated. The function ϕ is parameterized with the set of labels of an agent A (i.e. $\mathcal{L}(A)$) such that:

$$\forall\ l \in \mathcal{L}(A) \Longrightarrow \phi_{\mathcal{L}(A)}(l) = i \qquad i \in \{1, \ldots, card(\mathcal{L}(A))\}$$

The translation $\mathcal{C}[\![\ ]\!]_{\phi}^{\mathcal{F}\pi}$ generates a mini π-process that consists of three parts: i) the restrictions for the mapped labels $(_{\forall\ l\in\mathcal{L}(A)}\ \nu\ n_{\phi_{\mathcal{L}(A)}(l)})$ generates a restricted name for all labels in $\mathcal{L}(A)$ (note that $\phi_{\mathcal{L}(A)}(l)$ assigns a label l to a unique integer $i \in I\!N$), ii) the label

$$
\begin{array}{rcl}
\mathcal{C}[\![A]\!]^{\mathcal{F}\pi}_{\phi} & = & (\nu\ Error)(_{\forall\ l\in\mathcal{L}(A)}\ \nu\ n_{\phi_{\mathcal{L}(A)}(l)}) \\
& & (!Error(\_).\mathbf{0} \mid \prod_{l\in\mathcal{L}(A)} !n_{\phi_{\mathcal{L}(A)}(l)}(x_1,\ldots,x_{card(\mathcal{L}(A))}).\overline{x_{\phi_{\mathcal{L}(A)}(l)}}\langle\rangle \mid \mathcal{C}_A[\![A]\!]^{\mathcal{F}\pi}_{\phi}) \\
\mathcal{C}_A[\![\mathbf{0}]\!]^{\mathcal{F}\pi}_{\phi} & = & \mathbf{0} \\
\mathcal{C}_A[\![A_1 \mid A_2]\!]^{\mathcal{F}\pi}_{\phi} & = & \mathcal{C}_A[\![A_1]\!]^{\mathcal{F}\pi}_{\phi} \mid \mathcal{C}_A[\![A_2]\!]^{\mathcal{F}\pi}_{\phi} \\
\mathcal{C}_A[\![(\nu\ a)A]\!]^{\mathcal{F}\pi}_{\phi} & = & (\nu\ a)\mathcal{C}_A[\![A]\!]^{\mathcal{F}\pi}_{\phi} \\
\mathcal{C}_A[\![a(X).A]\!]^{\mathcal{F}\pi}_{\phi} & = & a(X).\mathcal{C}_A[\![A]\!]^{\mathcal{F}\pi}_{\phi} \\
\mathcal{C}_A[\![Y_l(X).A]\!]^{\mathcal{F}\pi}_{\phi} & = & (\nu\ r)(\overline{Y}\langle l,r,Error\rangle \mid r(u_l).u_l(X).\mathcal{C}_A[\![A]\!]^{\mathcal{F}\pi}_{\phi} \\
\mathcal{C}_A[\![!V(X).A]\!]^{\mathcal{F}\pi}_{\phi} & = & !\mathcal{C}_A[\![V(X).A]\!]^{\mathcal{F}\pi}_{\phi} \\
\mathcal{C}_A[\![\overline{a}(F)]\!]^{\mathcal{F}\pi}_{\phi} & = & (\nu\ f)(\overline{a}\langle f\rangle \mid !\mathcal{C}^{\mathcal{L}(A)}_{f}[\![F]\!]^{\mathcal{F}\pi}_{\phi}) \\
\mathcal{C}_A[\![\overline{Y_l}(F)]\!]^{\mathcal{F}\pi}_{\phi} & = & (\nu\ r)(\overline{Y}\langle l,r,Error\rangle \mid r(u_l).(\nu\ f)(\overline{u_l}\langle f\rangle \mid !\mathcal{C}^{\mathcal{L}(A)}_{f}[\![F]\!]^{\mathcal{F}\pi}_{\phi})) \\
\mathcal{C}_A[\![[F\leftarrow l]\ A_1\ A_2]\!]^{\mathcal{F}\pi}_{\phi} & = & (\nu\ r,r')(\mathcal{C}^{\mathcal{L}(A)}_{f}[\![F]\!]^{\mathcal{F}\pi}_{\phi} \mid \overline{f}\langle l,r,r'\rangle \\
& & \mid r(\_).\mathcal{C}_A[\![A_1]\!]^{\mathcal{F}\pi}_{\phi} \mid r'(\_).\mathcal{C}_A[\![A_2]\!]^{\mathcal{F}\pi}_{\phi})
\end{array}
$$

Table 7.6: Encoding of the FORM calculus in the mini π-calculus (part I).

processes, and iii) the encoded agent. The function $\mathcal{C}[\![\ ]\!]^{\mathcal{F}\pi}_{\phi}$ uses three subfunctions: i) $\mathcal{C}_A[\![\ ]\!]^{\mathcal{F}\pi}_{\phi}$ encodes an agent, ii) $\mathcal{C}^{\mathcal{L}(A)}_{f}[\![\ ]\!]^{\mathcal{F}\pi}_{\phi}$ encodes a form into a mini π-process located at channel f, and iii) $\mathcal{C}^{\mathcal{L}(A)}_{label}[\![\ ]\!]^{\mathcal{F}\pi}_{\phi}$ generates the test output-particle for the actual form (refer to Tables 7.6 and 7.7).

The function $\mathcal{C}[\![\ ]\!]^{\mathcal{F}\pi}$ is used to map transitions of the FORM calculus to mini π-actions. It is important to note that this mapping does not preserve the structure of an action. For example, both $\overline{a}(\langle\widetilde{l=b}\rangle)$ and $(\nu\ \tilde{c})\overline{a}(\langle\widetilde{l=b}\rangle)$ are mapped to $(\nu\ f)\overline{a}(f)$. The reason is that the encoding $\mathcal{C}[\![\ ]\!]^{\mathcal{F}\pi}_{\phi}$ adds one level of indirection for the communication of forms. More precisely, a form is translated into a replicated process located at a channel f. We can think of this channel as a "pointer" to a record. In order to access a value of the record, it is necessary to send a selector (the channel that maps the corresponding label) along the channel f.

The reader should note that the translation $\mathcal{C}[\![\ ]\!]^{\mathcal{F}\pi}_{\phi}$ is only defined on closed agents. This constraint is due to the definition of the weak $\mathcal{F}$-bisimulation which is only defined on closed agents.

$$
\begin{array}{rcl}
\mathcal{C}_f^{\mathcal{L}(A)}[\![\langle\rangle]\!]_\phi^{\mathcal{F}\pi} & = & f(L,R,E).\overline{E}\langle\rangle \\
\mathcal{C}_f^{\mathcal{L}(A)}[\![X]\!]_\phi^{\mathcal{F}\pi} & = & f(L,R,E).\overline{X}\langle L,R,E\rangle \\
\mathcal{C}_f^{\mathcal{L}(A)}[\![F\langle l{=}a\rangle]\!]_\phi^{\mathcal{F}\pi} & = & f(L,R,E).(\nu\ l,c)(\ \mathcal{C}_{label}^{\mathcal{L}(A)}[\![l,c]\!]_\phi^{\mathcal{F}\pi} \\
 & & \qquad\qquad\quad \mid l(\_).\overline{R}\langle a\rangle \\
 & & \qquad\qquad\quad \mid c(\_).(\nu\ f')(\overline{f'}\langle L,R,E\rangle \mid \mathcal{C}_{f'}^{\mathcal{L}(A)}[\![F]\!]_\phi^{\mathcal{F}\pi})) \\
\mathcal{C}_f^{\mathcal{L}(A)}[\![F\langle l{=}\mathcal{E}\rangle]\!]_\phi^{\mathcal{F}\pi} & = & f(L,R,E).(\nu\ l,c)(\ \mathcal{C}_{label}^{\mathcal{L}(A)}[\![l,c]\!]_\phi^{\mathcal{F}\pi} \\
 & & \qquad\qquad\quad \mid l(\_).\overline{E}\langle\rangle \\
 & & \qquad\qquad\quad \mid c(\_).(\nu\ f')(\overline{f'}\langle L,R,E\rangle \mid \mathcal{C}_{f'}^{\mathcal{L}(A)}[\![F]\!]_\phi^{\mathcal{F}\pi})) \\
\mathcal{C}_f^{\mathcal{L}(A)}[\![F\langle l{=}Y_k\rangle]\!]_\phi^{\mathcal{F}\pi} & = & f(L,R,E).(\nu\ l,c)(\ \mathcal{C}_{label}^{\mathcal{L}(A)}[\![l,c]\!]_\phi^{\mathcal{F}\pi} \\
 & & \qquad\qquad\quad \mid l(\_).(\nu\ r)(\overline{Y}\langle k,r,E\rangle \mid r(u_k).\overline{R}\langle u_k\rangle) \\
 & & \qquad\qquad\quad \mid c(\_).(\nu\ f')(\overline{f'}\langle L,R,E\rangle \mid \mathcal{C}_{f'}^{\mathcal{L}(A)}[\![F]\!]_\phi^{\mathcal{F}\pi})\ ) \\
\mathcal{C}_f^{\mathcal{L}(A)}[\![F{\cdot}X]\!]_\phi^{\mathcal{F}\pi} & = & f(L,R,E).(\nu\ c)(\ \overline{X}\langle L,R,c\rangle \\
 & & \qquad\qquad \mid c(\_).(\nu\ f')(\overline{f'}\langle L,R,E\rangle \mid \mathcal{C}_{f'}^{\mathcal{L}(A)}[\![F]\!]_\phi^{\mathcal{F}\pi})) \\
\mathcal{C}_f^{\mathcal{L}(A)}[\![F\backslash X]\!]_\phi^{\mathcal{F}\pi} & = & f(L,R,E).(\nu\ c)(\overline{X}\langle L,E,c\rangle \\
 & & \qquad\qquad \mid c(\_).(\nu\ f')(\overline{f'}\langle L,R,E\rangle \mid \mathcal{C}_{f'}^{\mathcal{L}(A)}[\![F]\!]_\phi^{\mathcal{F}\pi})) \\
 & & \\
\mathcal{C}_{label}^{\mathcal{L}(A)}[\![l,c]\!]_\phi^{\mathcal{F}\pi} & = & \overline{L}\langle x_1,\ldots,x_{j-1},x_j,x_{j+1},\ldots,x_n\rangle, \\
 & & \text{with}\ \ x_j = l;\ x_1,\ldots,x_{j-1},x_{j+1},\ldots,x_n = c \\
 & & \text{and}\ \ \ \phi_{\mathcal{L}(A)}(l) = j \\
 & & \\
\mathcal{C}[\![\tau]\!]^{\mathcal{F}\pi} & = & \tau \\
\mathcal{C}[\![a(\langle\widetilde{l{=}b}\rangle)]\!]^{\mathcal{F}\pi} & = & a(f) \\
\mathcal{C}[\![\overline{a}(\langle\widetilde{l{=}b}\rangle)]\!]^{\mathcal{F}\pi} & = & (\nu\ f)\overline{a}\langle f\rangle \\
\mathcal{C}[\![(\nu\ \tilde{c})\overline{a}(\langle\widetilde{l{=}b}\rangle)]\!]^{\mathcal{F}\pi} & = & (\nu\ f)\overline{a}\langle f\rangle
\end{array}
$$

Table 7.7: Encoding of the FORM calculus in the mini π-calculus (part II).

In the following, we show that the encoding of the FORM calculus in the mini π-calculus is faithful: if two agents A_1 and A_2 are weakly $\mathcal{F}$-bisimular (i.e. $A_1 \overset{\mathcal{F}}{\approx} A_2$), then it also holds that $\mathcal{C}[\![A_1]\!]_\phi^{\mathcal{F}\pi} \approx \mathcal{C}[\![A_2]\!]_\phi^{\mathcal{F}\pi}$.

Lemma 7.2 *Let A be a closed agent and μ be an action in the* FORM *calculus. Then if $A \overset{\mu}{\Longrightarrow} A'$, then there exists a mini π-process P such that $\mathcal{C}[\![A]\!]_\phi^{\mathcal{F}\pi} \overset{\mathcal{C}[\![\mu]\!]^{\mathcal{F}\pi}}{\Longrightarrow} P$.*

PROOF: We proceed by induction on the structure of A with $A \not\equiv \mathbf{0}$. By assumption the agent A is closed. Therefore, the outermost subject part of A must be a simple name and the outermost object part does not contain unbound form variables. Only the most significant cases are considered.

- $A = a(X).A_1$.

  We have $\mathcal{C}[\![A]\!]_\phi^{\mathcal{F}\pi} = a(X).\mathcal{C}[\![A_1]\!]_\phi^{\mathcal{F}\pi}$. It holds that $a(X).A_1 \overset{a(\langle \widetilde{l=b} \rangle)}{\Longrightarrow} A_1\{\langle \widetilde{l=b} \rangle / X\}$ and $\mathcal{C}[\![A]\!]_\phi^{\mathcal{F}\pi} \overset{a(f)}{\Longrightarrow} P$ with $P = \mathcal{C}[\![A_1]\!]_\phi^{\mathcal{F}\pi}\{f/X\}$.

- $A = \overline{a}(\langle \widetilde{l=b} \rangle)$.

  We have $\mathcal{C}[\![A]\!]_\phi^{\mathcal{F}\pi} = (\nu\ f)(\overline{a}\langle f \rangle \mid !\mathcal{C}_f^{\mathcal{L}(A)}[\![\langle \widetilde{l=b} \rangle]\!]_\phi^{\mathcal{F}\pi})$. It holds that $\overline{a}(\langle \widetilde{l=b} \rangle) \overset{\overline{a}(\langle \widetilde{l=b} \rangle)}{\Longrightarrow} \mathbf{0}$ and $\mathcal{C}[\![A]\!]_\phi^{\mathcal{F}\pi} \overset{(\nu\ f)\overline{a}\langle f \rangle}{\Longrightarrow} P$ with $P = \mathbf{0} \mid !\mathcal{C}_f^{\mathcal{L}(A)}[\![\langle \widetilde{l=b} \rangle]\!]_\phi^{\mathcal{F}\pi}$. □

Lemma 7.3 *Let A be a closed agent, μ an action, and P a mini π-process. Then $\mathcal{C}[\![A]\!]_\phi^{\mathcal{F}\pi} \overset{\mathcal{C}[\![\mu]\!]^{\mathcal{F}\pi}}{\Longrightarrow} P$ implies that there exists an agent A' such that $A \overset{\mu}{\Longrightarrow} A'$.*

PROOF: We proceed by induction on the structure of A with $A \not\equiv \mathbf{0}$. By assumption the agent A is closed. Therefore, the outermost subject part of A must be a simple name and the outermost object part does not contain unbound form variables. Only the most significant cases are considered.

- $A = a(X).A_1$.

  We have $\mathcal{C}[\![A]\!]_\phi^{\mathcal{F}\pi} = a(X).\mathcal{C}[\![A_1]\!]_\phi^{\mathcal{F}\pi}$. It holds that $\mathcal{C}[\![A]\!]_\phi^{\mathcal{F}\pi} \overset{\mathcal{C}[\![a(\langle \widetilde{l=b} \rangle)]\!]^{\mathcal{F}\pi}}{\Longrightarrow} P$ with $P = \mathcal{C}[\![A_1]\!]_\phi^{\mathcal{F}\pi}\{f/X\}$, $a(X).A_1 \overset{a(\langle \widetilde{l=b} \rangle)}{\Longrightarrow} A'$ with $A' = A_1\{\langle \widetilde{l=b} \rangle / X\}$, and $a(f) = \mathcal{C}[\![a(\langle \widetilde{l=b} \rangle)]\!]^{\mathcal{F}\pi}$.

- $A = \overline{a}(\langle \widetilde{l=b} \rangle)$.

  We have $\mathcal{C}[\![A]\!]_\phi^{\mathcal{F}\pi} = (\nu\ f)(\overline{a}\langle f \rangle \mid !\mathcal{C}_f^{\mathcal{L}(A)}[\![\langle \widetilde{l=b} \rangle]\!]_\phi^{\mathcal{F}\pi})$. It holds that $\mathcal{C}[\![A]\!]_\phi^{\mathcal{F}\pi} \overset{\mathcal{C}[\![\overline{a}(\langle \widetilde{l=b} \rangle)]\!]^{\mathcal{F}\pi}}{\Longrightarrow} P$ with $P = \mathbf{0} \mid !\mathcal{C}_f^{\mathcal{L}(A)}[\![\langle \widetilde{l=b} \rangle]\!]_\phi^{\mathcal{F}\pi}$, $\overline{a}(\langle \widetilde{l=b} \rangle) \overset{\overline{a}(\langle \widetilde{l=b} \rangle)}{\Longrightarrow} A'$ with $A' = \mathbf{0}$, and $\mathcal{C}[\![\overline{a}(\langle \widetilde{l=b} \rangle)]\!]^{\mathcal{F}\pi} = (\nu\ f)\overline{a}\langle f \rangle$. □

Lemma 7.4 *Let A be an agent, $X \in \mathrm{fv}(A)$, X_l a subterm of A, and b a subject of an action μ. Then if $\mathcal{C}[\![A\{\langle l\!=\!b\rangle/X\}]\!]_\phi^{\mathcal{F}\pi} \overset{\mathcal{C}[\![\mu]\!]^{\mathcal{F}\pi}}{\Longrightarrow} \mathcal{C}[\![A'\{\langle l\!=\!b\rangle/X\}]\!]_\phi^{\mathcal{F}\pi}$ it holds that*

$$\mathcal{C}[\![A]\!]_\phi^{\mathcal{F}\pi}\{f/X\} \overset{\tau}{\Longrightarrow} \cdot \xrightarrow{(\nu\ r)\overline{f'}\langle l,r,Error\rangle} \cdot \xrightarrow{r(b)} \cdot \xrightarrow{\mathcal{C}[\![\mu]\!]^{\mathcal{F}\pi}} \cdot \overset{\tau}{\Longrightarrow} \mathcal{C}[\![A']\!]_\phi^{\mathcal{F}\pi}\{f/X\}$$

with f being the location of $\mathcal{C}_f^{\mathcal{L}(A)}[\![X]\!]_\phi^{\mathcal{F}\pi}$.

PROOF: By assumption it holds that

$$\mathcal{C}[\![A\{\langle l\!=\!b\rangle/X\}]\!]_\phi^{\mathcal{F}\pi} \overset{\tau}{\Longrightarrow} \cdot \xrightarrow{\mathcal{C}[\![\mu]\!]^{\mathcal{F}\pi}} \cdot \overset{\tau}{\Longrightarrow} \mathcal{C}[\![A'\{\langle l\!=\!b\rangle/X\}]\!]_\phi^{\mathcal{F}\pi}.$$

A projection X_l is encoded as the mini π-calculus fragment

$$(\nu\ r)(\overline{X}\langle l, r, Error\rangle \mid r(u_l).P$$

where P is a mini π-process that uses u_l. Applying $\{f/X\}$ to this fragment leads to

$$(\nu\ r)(\overline{f}\langle l, r, Error\rangle \mid r(u_l).P).$$

In order the perform an action along the channel b (with is denoted by X_l), the encoding of the agent $A\{\langle l\!=\!b\rangle/X\}$ performs two communications with

$$\xrightarrow{(\nu\ r)\overline{f'}\langle l,r,Error\rangle} \cdot \xrightarrow{r(b)} \cdot \xrightarrow{\mathcal{C}[\![\mu]\!]^{\mathcal{F}\pi}}$$

as required. □

Theorem 7.2 *For each pair of closed agents A_1 and A_2, $\mathcal{C}[\![\ ]\!]_\phi^{\mathcal{F}\pi}$ is an injective mapping, such that if $A_1 \overset{\mathcal{F}}{\approx} A_2$, then it holds that*

$$\mathcal{C}[\![A_1]\!]_\phi^{\mathcal{F}\pi} \ \approx\ \mathcal{C}[\![A_2]\!]_\phi^{\mathcal{F}\pi}.$$

PROOF: We show that the relation

$$\mathcal{R} = \{\ (\mathcal{C}[\![A_1]\!]_\phi^{\mathcal{F}\pi}, \mathcal{C}[\![A_2]\!]_\phi^{\mathcal{F}\pi})\ \mid\ A_1\ \overset{\mathcal{F}}{\approx} A_2\ \} \cup\ \approx$$

is a weak 1-bisimulation [ACS96].

Consider the case τ or *output actions*:

> Suppose $\mathrm{bn}(\mathcal{C}[\![\mu]\!]^{\mathcal{F}\pi}) \cup \mathrm{fn}(\mathcal{C}[\![A_1]\!]_\phi^{\mathcal{F}\pi} \mid \mathcal{C}[\![A_2]\!]_\phi^{\mathcal{F}\pi}) = \emptyset$. This condition can easily be established by α-converting the encoding of the action μ (i.e. $\mathcal{C}[\![\mu]\!]^{\mathcal{F}\pi}$), since the names of the bound names in the action are not important as long as they denote the same location. Let $\mathcal{C}[\![A_1]\!]_\phi^{\mathcal{F}\pi} \overset{\mathcal{C}[\![\mu]\!]^{\mathcal{F}\pi}}{\Longrightarrow} P$. By Lemma 7.3 it follows that there exists an agent A_1' such that $A_1 \overset{\mu}{\Longrightarrow} A_1'$.

The definition of weak $\mathcal{F}$-bisimulation guarantees that there exists an agent A_2' such that $A_2 \stackrel{\mu}{\Longrightarrow} A_2'$ and $A_1' \stackrel{\mathcal{F}}{\approx} A_2'$. By Lemma 7.2 it follows that $\mathcal{C}[\![A_2]\!]_\phi^{\mathcal{F}\pi} \stackrel{\mathcal{C}[\![\mu]\!]^{\mathcal{F}\pi}}{\Longrightarrow} P'$. Therefore, we have $P \; \mathcal{R} \; P'$ (as desired).

In the case of an output action, we know by Lemma 7.4 that the next weak transition $\mathcal{C}[\![\mu]\!]^{\mathcal{F}\pi}$ cannot occur before all names which are part of μ have been communicated. For each name, there are two communications. Since $P \; \mathcal{R} \; P'$, it follows that both communications must perform the same name requests.

Consider the case of composition with output:

Since $A_1 \stackrel{\mathcal{F}}{\approx} A_2$, we have $\mathcal{C}[\![A_1]\!]_\phi^{\mathcal{F}\pi} \approx \mathcal{C}[\![A_2]\!]_\phi^{\mathcal{F}\pi}$. Then for all messages $\overline{a}\langle\tilde{b}\rangle$ we have by definition $(\mathcal{C}[\![A_1]\!]_\phi^{\mathcal{F}\pi} \mid \overline{a}\langle\tilde{b}\rangle, \mathcal{C}[\![A_2]\!]_\phi^{\mathcal{F}\pi} \mid \overline{a}\langle\tilde{b}\rangle) \in \mathcal{R}$. $\square$

Theorem 7.3 *For each pair of closed agents A_1 and A_2, if $\mathcal{C}[\![A_1]\!]_\phi^{\mathcal{F}\pi} \approx \mathcal{C}[\![A_2]\!]_\phi^{\mathcal{F}\pi}$, then it holds that $A_1 \stackrel{\mathcal{F}}{\approx} A_2$.*

PROOF: We show that the relation

$$\mathcal{R} = \{ \; (A_1, A_2) \;\mid\; \mathcal{C}[\![A_1]\!]_\phi^{\mathcal{F}\pi} \approx \mathcal{C}[\![A_2]\!]_\phi^{\mathcal{F}\pi} \; \} \cup \stackrel{\mathcal{F}}{\approx}$$

is a weak $\mathcal{F}$-bisimulation.
Consider the case τ or *output actions*:

Suppose $\mathrm{bn}(\mu) \cup \mathrm{fn}(A_1 \mid A_2) = \emptyset$. This condition can be easily established by α-converting the encoding of the action μ (i.e. $\mathcal{C}[\![\mu]\!]^{\mathcal{F}\pi}$), since the names of the bound names in the action are not important as long as they denote the same location. Let $A_1 \stackrel{\mu}{\Longrightarrow} A_1'$. By Lemma 7.2 there exists a mini π-process P_1 such that $\mathcal{C}[\![A_1]\!]_\phi^{\mathcal{F}\pi} \stackrel{\mathcal{C}[\![\mu]\!]^{\mathcal{F}\pi}}{\Longrightarrow} P_1$.

The definition of weak 1-bisimulation guarantees that there exists a process P_2 such that $\mathcal{C}[\![A_2]\!]_\phi^{\mathcal{F}\pi} \stackrel{\mathcal{C}[\![\mu]\!]^{\mathcal{F}\pi}}{\Longrightarrow} P_2$ and $P_1 \approx P_2$. By Lemma 7.3 it immediately follows that $A_2 \stackrel{\mu}{\Longrightarrow} A_2'$, which implies that $A_1' \; \mathcal{R} \; A_2'$.

In the case of an output action, Lemma 7.4 shows that the next weak transition $\mathcal{C}[\![\mu]\!]^{\mathcal{F}\pi}$ cannot occur before all names which are part of μ have been communicated. For each name, there are two communications. Since $P_1 \approx P_2$, it follows that both processes must perform the same name requests. Furthermore, P_1 and P_2 perform more moves than A_1' and A_2', but if the next move for P_1 and P_2 is $\mathcal{C}[\![\mu]\!]^{\mathcal{F}\pi}$, then A_1 and A_2 can move for μ, too.

Consider now the case of composition with output:

Since $\mathcal{C}[\![A_1]\!]_\phi^{\mathcal{F}\pi} \approx \mathcal{C}[\![A_2]\!]_\phi^{\mathcal{F}\pi}$, it follows that $A_1 \stackrel{\mathcal{F}}{\approx} A_2$. Then for all messages $\overline{a}(\langle \widetilde{l\!=\!b} \rangle)$ we have by definition $((A_1 \mid \overline{a}(\langle \widetilde{l\!=\!b} \rangle)), (A_2 \mid \overline{a}(\langle \widetilde{l\!=\!b} \rangle)) \in \mathcal{R}$. $\square$

The faithfulness of the encoding $\mathcal{C}[\![\ ]\!]_{\phi}^{\mathcal{F}\pi}$ follows from the Theorems 7.2 and 7.3.

The reader should note that the encoding of the FORM calculus in the mini π-calculus and vice versa allows us to define corresponding (faithful) encodings between the FORM calculus and the $\pi\mathcal{L}$-calculus:

$$\begin{array}{rcl} \mathcal{C}[\![\ ]\!]^{\mathcal{FL}} & = & \mathcal{C}[\![\ ]\!]^{\pi\mathcal{L}} \circ \mathcal{C}[\![\ ]\!]^{\mathcal{F}\pi} \\ \mathcal{C}[\![\ ]\!]^{\mathcal{LF}} & = & \mathcal{C}[\![\ ]\!]^{\pi\mathcal{F}} \circ \mathcal{C}[\![\ ]\!]^{\mathcal{L}\pi} \end{array}$$

It is easy to show that both encodings $\mathcal{C}[\![\ ]\!]^{\mathcal{FL}}$ and $\mathcal{C}[\![\ ]\!]^{\mathcal{LF}}$ are faithful: if two agents A_1 and A_2 are weakly $\mathcal{F}$-bisimular, then it holds that $P_1 = \mathcal{C}[\![A_1]\!]_{\phi}^{\mathcal{F}\pi}$ and $P_2 = \mathcal{C}[\![A_2]\!]_{\phi}^{\mathcal{F}\pi}$ are weakly 1-bisimular (i.e. $P_1 \approx P_2$). Furthermore, due to the fact that the encoding $\mathcal{C}[\![\ ]\!]^{\pi\mathcal{L}}$ is faithful, it also holds that $A_1' = \mathcal{C}[\![P_1]\!]^{\pi\mathcal{L}}$ and $A_2' = \mathcal{C}[\![P_2]\!]^{\pi\mathcal{L}}$ are weakly $\mathcal{L}$-bisimular. Hence, it immediately follows that $\mathcal{C}[\![A_1]\!]^{\mathcal{FL}}$ and $\mathcal{C}[\![A_1]\!]^{\mathcal{FL}}$ are weakly $\mathcal{L}$-bisimular, which proofs the faithfulness of $\mathcal{C}[\![\ ]\!]^{\mathcal{FL}}$. The corresponding proof for $\mathcal{C}[\![\ ]\!]^{\mathcal{LF}}$ is similar.

The question arises whether it is not easier to define the encodings between the FORM calculus and the $\pi\mathcal{L}$-calculus directly (instead of using the corresponding encodings in the mini π-calculus).

A direct encoding of the FORM calculus in the $\pi\mathcal{L}$-calculus is not straightforward. In particular, the encoding of matching requires some reflection. Due to the fact that in the $\pi\mathcal{L}$-calculus it is not possible to check whether a given form F defines a binding for a label l (if a form does not define a binding for a label l, a projection results in the empty binding $\mathcal{E}$, which does not allow for any further analysis), forms of the FORM calculus cannot be simply encoded as forms in the $\pi\mathcal{L}$-calculus. Again, a strategy has to be used to encode forms as agents (similar to the corresponding encoding in the mini π-calculus).

As we will illustrate in section 7.4, the FORM calculus is a not a superset of the $\pi\mathcal{L}$-calculus, since both calculi differ in the way polymorphic form extension is defined. A direct encoding of the $\pi\mathcal{L}$-calculus in the FORM calculus has to consider the different semantics of polymorphic extension and, therefore, is not straightforward, either.

7.4 Comparison of the FORM calculus with the $\pi\mathcal{L}$-calculus

Although the FORM calculus is derived from the $\pi\mathcal{L}$-calculus, there are, nevertheless, several differences between the two calculi. In this section, we discuss the main differences, particularly focusing on polymorphic form extension.

As the main difference between the two calculi, consider i) the extension of the syntax for forms with the concept of polymorphic form restriction and ii) the introduction of label matching in agent expressions: the former allows a user to remove a set of labels of a given form whereas the latter is a mechanism to check for the name bound by a label in a form. Both concepts have been extensively discussed in section 7.3. Furthermore, we have separated the notations for empty form $\langle\rangle$ and empty binding $\mathcal{E}$ (which are merged in the $\pi\mathcal{L}$-calculus), and assigned the extension of a form with an empty binding a special meaning in order to define polymorphic form restriction in a more natural way.

Both calculi define the concept of polymorphic form extension, but as we will illustrate below, the semantics differs in the context of extensions with an empty binding. In order to illustrate this fact, consider the two forms $F = \langle\rangle\langle l = a\rangle\langle m = b\rangle$ and $G = \langle\rangle\langle l = c\rangle\langle m = \mathcal{E}\rangle$. In the $\pi\mathcal{L}$-calculus, a polymorphic extension of F with G leads to the form $F\ G = \langle\rangle\langle l = a\rangle\langle m = b\rangle\langle l = c\rangle\langle m = \mathcal{E}\rangle$. Hence, $(F\ G)_l = c$, $(F\ G)_m = \mathcal{E}$, and by definition of the equivalence of forms, $F\ G \equiv \langle\rangle\langle l = c\rangle$. In the FORM calculus, however, $(F{\cdot}G)_l = c$ and $(F{\cdot}G)_m = b$, which implies that $F{\cdot}G \equiv \langle\rangle\langle m = b\rangle\langle l = c\rangle$. Therefore, $F\ G \not\equiv F{\cdot}G$.

From a different perspective, polymorphic form extension in the $\pi\mathcal{L}$-calculus can be seen as a simple record concatenation (i.e. the label bindings of the right-hand side form are simply added to the left-hand side form) whereas in the FORM calculus, polymorphic form extension *removes* extensions with an empty binding (and the corresponding overridden bindings) *before* the set of labels are merged. This also explains why we have adapted a different notation for polymorphic form extension in the FORM calculus as is used in the $\pi\mathcal{L}$-calculus.

As a summary of this section, we can say that the definition of the FORM calculus cannot be considered as a simple extension of the $\pi\mathcal{L}$-calculus (or as a a "layer of syntactic sugar" on top of $\pi\mathcal{L}$), as i) the semantics of polymorphic form extension is different and ii) the FORM calculus defines abstractions which cannot be encoded in the $\pi\mathcal{L}$-calculus in a straightforward way (refer to the discussion about matching at the end of section 7.3.11).

Chapter 8

Modelling objects in the FORM calculus

Concurrent programming demands special languages that provide primitives for communication and synchronization. Using objects, it turns out that a general programming model can easily be made concurrent and parallel, and several authors have shown that the π-calculus (or some variant) has enough expressive power to model standard features of concurrent, object-oriented programming languages in a convenient way [Jon93, Vas94, BS95, Wal95, San96b].

Many of the formal models mentioned above only support a subset of common features found in object-oriented languages, and the resulting abstractions suffer from fixed, positional tuple-based interfaces. Furthermore, none of the models allows us to explore the common behaviour and protocols of classes and objects, and each object has to be defined from scratch. As an approach to overcome these problems, we define a meta-level framework for concurrent, object-oriented programming and show that many concepts of concurrent, object-oriented languages such as encapsulation, self-references, synchronization, dynamic binding, classes, mixins, inheritance etc. can be conveniently modelled with the aid of agents representing meta-level abstractions.

Forms, polymorphic extension, and polymorphic restriction, we argue, are the key mechanisms for extensibility, flexibility, and robustness of the meta-level framework. In particular, polymorphic extension and restriction in combination with keyword-based parameters facilitate the definition of object-oriented abstractions as they enable the definition of an extensible and adaptable meta-level framework for modelling concurrent, object-oriented features. Furthermore, by using a process calculus, concurrency does not need to be modelled explicitly, and we can focus on the definition of extensible, higher-level abstractions.

The specification of the meta-level framework does not only allow us to define the semantics of various object-oriented abstractions in terms of a small set of primitives, but also to illustrate the expressive power of the underlying calculus, in particular the resulting extensibility, flexibility, and robustness.

In order to explain our modelling steps at a higher level of abstraction than the (rather low-level) FORM calculus, we will use PICCOLA(F), an experimental composition language currently under development at our institute, as an executable specification language. Like for the PICT programming language [PT97], the language features of PICCOLA(F) are all defined by transformation to a core language that implements the FORM calculus. Therefore, it is as much an attempt to turn the FORM calculus into a full-blown programming language as it is a platform for experimenting with compositional abstractions and for modelling language features.

In fact, we are currently exploring several approaches in the development of a composition language called PICCOLA: an approach emphasizing a more functional and declarative style of programming [ALSN99] and another approach based on an imperative style of programming [Lum99]. Using such an approach, we hope to discover the right abstractions for software composition and to define an unified paradigm which fulfills the requirements given in section 2.5. Common to both approaches mentioned above is the fact that all language features are defined by transformation to a core language that implements the $\pi\mathcal{L}$-calculus. In order to distinguish between the different working versions of PICCOLA, we will use PICCOLA(F) in the following to denote the variant based on the FORM calculus which emphasizes an imperative style of programming.

A detailed description of PICCOLA(F) is beyond the scope of this work; we will only explain particular abstractions as we need them (see appendix C for the PICCOLA(F) language definition). For additional information about the versions based on the $\pi\mathcal{L}$-calculus, refer to [ALSN99] and [Lum99].

This chapter is organized as follows: we first introduce the basic object model of Pierce and Turner which forms the basis our models. We extend this model with common features of object-oriented programming languages and abstractions for higher-level synchronization. We continue with a discussion of class abstractions for different flavours of inheritance and method dispatch and introduce models for mixins and mixin-based inheritance. We identify the key concepts for extension and generalization of the resulting model and define a meta-level framework for modelling concurrent object-oriented programming abstractions. We briefly point out the limits of the FORM calculus by illustrating object-oriented abstractions that cannot be encoded in the calculus itself. Finally, we conclude with a discussion of related work, a comparison of other meta-level approaches, and a summary of the main observations of modelling objects in the FORM calculus.

8.1 Basic object model in the FORM calculus

In this section, we outline a basic object model in the FORM calculus as an extension of a record-based object model developed by Pierce and Turner [PT95]. Starting with an encoding of the Pierce and Turner basic object model in the FORM calculus, we extend the model with common features of object-oriented programming languages such as classes, dynamic binding, and inheritance.

8.1.1 The Pierce/Turner basic object model

Using the pure π-calculus, objects can be viewed as groups of processes [BS95, Wal95, San96b]. There is, however, no way that one process can directly affect or refer to another process, and it can only send messages along some channels where, by convention, the other process listens. Similarly, referring to a group of processes means that we can send messages to a collection of channels where these processes are listening. An attractive notion to model such a facility is the standard notion of *records* that enable us to selectively address one member of a group by using its *name*. Viewing each individual channel as an explicitly named "service access point", we can bundle them together in a record that provides a well-defined interface for accessing the related services. Furthermore, this packaging gives rise to a *higher-order* style of programming with objects, since a complete interface of one object may be manipulated as a single value.

Pierce and Turner outlined a basic model for objects in PICT, where i) an object is a set of persistent processes representing instance variables and methods and where ii) the interface of an object is a represented as a record containing the channels of the visible features [PT95]. The following example presents a FORM calculus encoding of a *reference cell*[1] in the basic object model of Pierce and Turner. This encoding shows that the FORM calculus provides a compact formalism for encoding record-based object models.

$$
\begin{array}{lcl}
\mathbf{def}\ \mathit{RefCell}(X) & = & (\nu\ \mathit{contents},\ s,\ g) \\
& & (\ \overline{\mathit{contents}}(\langle \mathit{val} = X_{init}\rangle) \\
& & \mid\ !g(Y).\mathit{contents}(Z).(\overline{Y_{result}}(Z) \mid \overline{\mathit{contents}}(Z)) \\
& & \mid\ !s(Y).\mathit{contents}(\_).(\overline{Y_{result}}(\langle\rangle) \mid \overline{\mathit{contents}}(Z)) \\
& & \mid\ \overline{X_{result}}(\langle \mathit{set} = s\rangle\langle \mathit{get} = g\rangle)\ )
\end{array}
$$

The agent listening on channel $\mathit{RefCell}$ implements an *object generator* [Coo89] which yields a new object if a form containing at least a binding for $init$ and $result$ is sent along channel $\mathit{RefCell}$. Access to the new object is returned along the channel denoted by X_{result}. The form $\langle set = s\rangle\langle get = g\rangle$ implements the interface to the newly created object. The methods of the object are implemented by the agents listening on the channels s and g (representing the set and get methods, respectively). The agents sending/listening along the channel $contents$ implement the state of the object. The state and the method implementations are local to each object.

In the core language of PICCOLA(F), a factory for reference cell objects could thus be modelled as:

```
new ref
run ref?*(Args) do
    let
        new contents, getCH, setCH
    in
        contents!(<val = Args.init>)      {- Initialization -}
        |( getCH?*(X) do contents?(V) do contents!V | X.result!V )
        |( setCH?*(X) do contents?(_) do contents!X | X.result!<> )
        | Args.result!(<get = getCH, set = setCH>)
    end
```

This agent accepts an initial value and a reply address, and creates a new reference cell suitably initialized for each such request. Then, it returns a form containing the channels

[1] A reference cell is an updatable data-structure.

getCH and setCH bound to the labels get and set of the form. The methods set and get are called using the notation *obj*.set(<*form*>) and *obj*.get(), respectively. The library agent PrVal prints the value of a form with a binding for the label val.

```
let
   new res, ack, cont, sig
in
   ref!(<init=2,result=res>)
   | res?(cell) do cell.set(<val=3,result=ack>)
   | ack?(_) do cell.get(<result=cont>)
   | cont?(X) do PrVal(<X,result=sig>)
   | sig?(_) do null
end
```

The example above illustrates that programming in the core language of PICCOLA(F) is like programming in a concurrent assembler. However, the language defines some syntactic sugar which makes it easier to define factory objects:

```
function ref (Args) =
   let
      new contents
      run contents!(<val = Args.init>)

      procedure get(X) do contents?(V) do contents!V | X.result!V
      procedure set(X) do contents?(_) do contents!X | X.result!<>
   in
      < set = set, get = get >
   end
```

Note that get and set requests cannot interfere since the bodies of the two servers each grab the resources of the reference cell (i.e. the contents message) and release them when they are done. Pierce and Turner generalize this model to concurrent objects whose methods synchronize with respect to a shared lock [PT95].

Due to the fact that i) defining a procedure or function in the local scope of a factory agent and ii) assigning it to a label of a result form is heavily used, PICCOLA(F) defines another layer of syntactic sugar of so-called *active forms* which allows us to rewrite the reference cell example as follows:

```
function ref (Args) =
   let
      new contents
      run contents!(<val = Args.init>)
   in
      <
         procedure get(X) do contents?(V) do contents!V | X.result!V,
         procedure set(X) do contents?(_) do contents!X | X.result!<>
      >
   end
```

Throughout the rest of this work, we will use the notations introduced above interchangeably.

The essentials of concurrent objects are captured by the basic object model of Pierce and Turner: encapsulation, identity, persistence, instantiation, and synchronization. It is less clear whether the model can be easily extended to capture other common and more esoteric features of object-oriented programming languages. Basic features found in most of the better known languages include dynamic binding, inheritance, overriding, and class variables.

In the following, we explore the extension of the Pierce/Turner object model to some of these features, including synchronization abstractions based on generic synchronization policies [McH94]. Since a detailed description of all modelling steps is beyond the scope of this section, we will only outline the main results (refer to [SL96] or [Var96] for further details).

8.1.2 Modelling requirements

We express our modelling requirements by means of a hierarchy of classes (based on an example presented in [VLM96]) illustrated in Figure 8.1, which we will use throughout the rest of this chapter. The class Point is the root of the hierarchy. It contains two private instance variables x and y, two accessor methods X and Y, a method move which moves a point by a given offset (dx,dy), and a method double that doubles the values of the x and y coordinates. The class Point also contains a class variable Counter which counts the number of its instances and a class method NoOfPoints to access the Counter variable. A specialization of Point is the class HistoryPoint which extends the method move and prints the instance variables x and y each time move is invoked. The reader should note that the method move is invoked by the method double (using a self call), which is not overridden in the class HistoryPoint. Another direct specialization of Point is the class BoundedPoint. It ensures that the y coordinate of an instance never exceeds a given upper bound b. This bound is a constant in the class BoundedPoint, although this is not necessary. In order to illustrate an non-constant bound, the class LinearBoundedPoint specializes the class BoundedPoint by overriding the method bound in an appropriate way (i.e. bound checks if the y coordinate is smaller than the x coordinate).

The remaining classes are less trivial as they are specializations of more than one parent-class. The class HistoryBoundedPoint includes the functionality of the classes BoundedPoint and HistoryPoint. The instance variables x and y and the method move that occurred in the class Point should be shared in the class HistoryBoundedPoint whereas the two different specializations of the method move should be combined. Using C++ terminology, we consider the class Point as a *shared* ancestor of the class HistoryBoundedPoint.

The class DoubleBoundedPoint ensures that the upper bound of an instance never exceeds two upper bounds: a horizontal bound and a linear bound. Therefore, the class DoubleBoundedPoint needs two versions of the method bound, once as a constant, and once as a linear function. Consequently, the methods bound must be duplicated. The method move needs to be invoked twice: once with each bound version. Hence, the class DoubleBoundedPoint needs duplication of the attributes (instance variables and

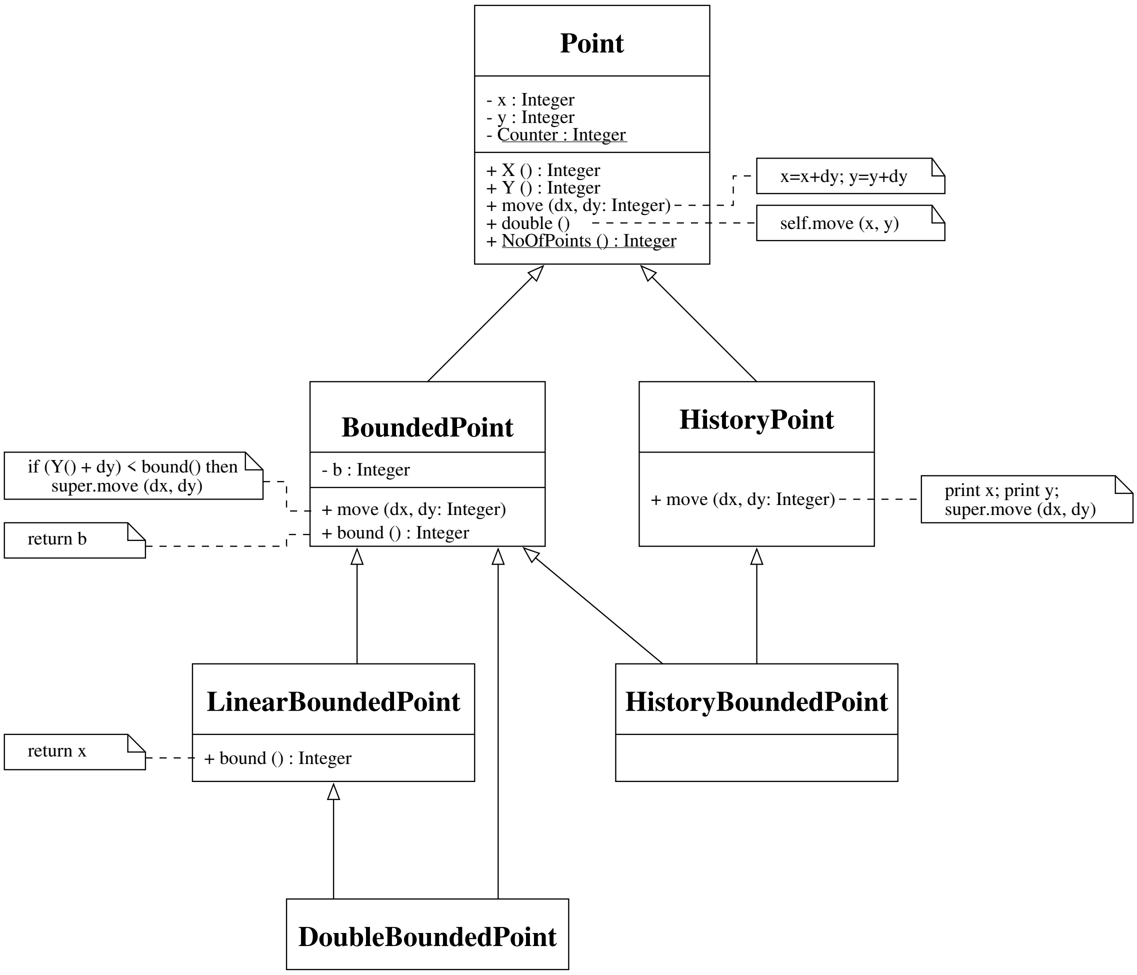


Figure 8.1: UML diagram of point class hierachy.

methods) introduced by the common ancestor BoundedPoint, while the attributes of the common ancestor Point need to be shared.

The following examples illustrate i) various degrees of encapsulation (i.e. object, class, and global scope), ii) static and dynamic binding (i.e. instance variables vs. overridden features), and iii) single inheritance.

8.1.3 Class and object features

Our first implementation of the class Point is a straightforward mapping into the Pierce/Turner basic object model (see Figure 8.2).

This first approach does not encapsulate common class features: class variables and methods are defined and implemented in the global scope. It is not possible to model self-references of objects, and there is no possibility to express the classes HistoryPoint or

```
new Counter                                       {- class variable -}
run Counter!<val = 0>

procedure NoOfPoints(P) do                        {- class method -}
      Counter?(X) do (Counter!X | P.result!X)

function PointClass(Init) =
  let
    new x                                         {- instance variables -}
    new y
    run (x!<val = Init.x> | y!<val = Init.y>)

    run Counter?(X) do Counter!<add(<lval = X.val, rval = 1>)>
  in
    <
      procedure X(Args) do x?(V) do (x!V | Args.result!V),
      procedure Y(Args) do y?(V) do (y!V | Args.result!V),
      procedure move(Args) do x?(VX) do y?(VY) do
             ( x!<add (<lval = VX.val, rval = Args.dx>)>
             | y!<add(<lval = VY.val, rval = Args.dy>)>
             | Args.result!<> ),
      procedure double(Args) do x?(VX) do y?(VY) do
             ( x!<add (<lval = VX.val, rval = VX.val>)>
             | y!<add(<lval = VY.val, rval = VY.val>)>
             | Args.result!<> )
    >
  end
```

Figure 8.2: Source code of first model in PICCOLA.

BoundedPoint as inheriting from Point. Furthermore, due to the lack of self-references, the encoding does not allow local method calls within an object: the method double cannot use the functionality of the method move and, therefore, has to reimplement the modification of the two coordinates in its body. On the other hand, the implementation of Point behaves correctly in the presence of concurrent clients: each method obtains and releases the necessary local resources in a consistent sequence, thus avoiding both interference and deadlock.

The reader should note that we have used a library abstraction `add` in order to add two integers.[2] Due to the fact the PICCOLA(F) version used for our models does not support infix operators (such as + or `*`), arithmetic operations have to be expressed as functions. In the case of summation, the function `add` requires bindings for the labels `lval` and `rval`, and returns its result (i.e. the sum of the values bound to `lval` and `rval`) in a form with a binding for `val`. Other arithmetic operators are implemented similarly. The function `cmp` is used for comparing integers. It returns -1, 0, or 1 if the value bound

[2]Like booleans, integers (and strings) can be encoded in the FORM calculus, but PICCOLA(F) provides some syntactic sugar in order to treat them as built-in data-types.

at the label `lval` is smaller than, equal to, or greater than the value bound at the label `rval`, respectively.

8.1.4 Dynamic binding and encapsulation

In order to encapsulate class variables, we use an approach to explicitly represent *classes as first-class objects*. Class metaobjects[3] turn out to be useful for i) the declaration, initialization, and access control of class variables, ii) the implementation of class methods, and iii) the creation and initialization of instances. Note that we have not introduced a metaobject protocol [KdRB91] to dynamically change the behaviour of classes at run-time, though the fact that we have explicit class metaobjects allows us to explore this possibility as well.

In Figure 8.3, we can see that the class Point is represented by a single instance of a class metaobject `PointClass`. The declaration

let value *var* = *form* **in** *body* **end**

is just an abbreviation for a communication that binds a form to some value:

let new `c` **in** `c!`*form* | `c?(`*var*`)` **do** *body* **end**

The class variable Counter is now represented as an instance variable of the class metaobject. Since there is no class metaobject factory (only a single instance), we ensure that all instances of the class Point will share the same, hidden variable `Counter`. Furthermore, the class method `Create` defines the *constructor* of the class Point. Note that throughout the rest of this work, a method `Create` will be used to denote the constructor of a given class.

Instead of representing instance variables as messages, they are now modelled as reference cells. This has the advantage of making them easier to use, but it means that we can no longer synchronize concurrent accesses. Various approaches are possible for avoiding interference and deadlock, and we will discuss an approach based on generic synchronization policies in section 8.2. For the rest of this section, we assume that clients' requests are sequential.

In the declaration part of `Create`, an *empty* reference cell (i.e. `Self`) and a new object (i.e. `NewInstance`) are created. In contrast to the reference cell introduced in section 8.1.1, an empty reference cell does not get an initial value at creation time. The value `NewInstance` is then assigned as the first (and only) contents of the `Self` cell and is returned as the result of `Create`. `Self` is used as an alias for `NewInstance` to realize self-reference. Since `Self` is declared before `NewInstance`, it can be used within the methods of `NewInstance`. The reader should note that the values `Self` and `NewInstance` form an abstraction comparable to an object generator with `Self`

[3]For the rest of this work, we use the same terminology for entities of the meta-level as is used by Kiczales et al. [KdRB91].

```
value PointClass =
  let
    value Counter = ref(<val = 0>)
  in
    <
      function NoOfPoints() = Counter.get(),
      function Create(Init) =
        let
          value Self = emptyRef()
          value NewInstance =
            let
              value x = ref(<val = Init.x>)       {- instance variables -}
              value y = ref(<val = Init.y>)
              function move(Args) =
                  x.set(<add(<lval=x.get().val,rval=Args.dx>)>);
                  y.set(<add(<lval=y.get().val,rval=Args.dy>)>)
            in
              Counter.set(<add (<lval=Counter.get().val, rval=1>)>);
              <
                move = move,
                function X() = x.get(),
                function Y() = y.get(),
                function double() =
                    Self.get().move(<dx=x.get().val,dy=y.get().val>)
              >
            end
        in
          Self.set(<NewInstance>);              {- self binding -}
          NewInstance
        end
    >
  end
```

Figure 8.3: Source code of `PointClass` class metaobject.

as formal parameter [Coo89] whereas `Self.set(...)` and `Self.get()` build a *fixed-point operator* [Red88]. The same encoding could also be used to obtain a self-reference for the class metaobject (not used in Figure 8.3). We will further discuss the notion of generators and fixed-point operators in section 8.3.

The class method `NoOfPoints` and the object constructor `Create` define the public interface of the class metaobject `PointClass`. Due to the fact that `Self` is visible in the context of all methods (`Self` is defined within the declaration part of the class method `Create`), there is no need to pass `Self` as parameter to every method. This explains why the methods `X`, `Y`, and `double` do not have a formal parameter.

In order to guarantee the correct behaviour in presence of concurrent clients, we did not support local methods nor local method calls in the previous model. However, in order to model different binding mechanisms for methods, we must introduce the notion of local methods and local method calls. The model presented in Figure 8.3 supports both static and dynamic binding of methods. If a method needs to be bound dynamically, the method has to be called by sending a message to `Self` (e.g. `Self.get().move(<...>)`). Static binding can be achieved by declaring a method within the scope of the object factory and calling it directly (e.g. `move(<...>)`. The reader should note that without additional concurrency control, the use of local method calls does not guarantee the correct synchronization of concurrent accesses.

A common form of a result in PICCOLA(F) is a continuation signal which indicates termination of an agent. Since continuation signals are frequently used to specify a sequence of operations, PICCOLA(F) provides a convenient syntax for their usage: if `v` is a value expression whose result is an empty form, then `v;A` is an agent that evaluates `v`, waits for its termination, and continues as `A`. Formally,

```
v ; A
```

is translated into

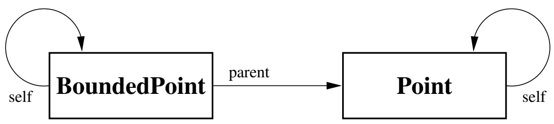
```
let new c in c!v | c?(_) do A end
```

Similarly, `v;` is an abbreviation for `v;null` (i.e. the **0** agent). Using sequencing of operations, it is possible to implement the methods of `PointClass` as *functions* (and not as procedures), which allows us to omit explicit `result` channels.

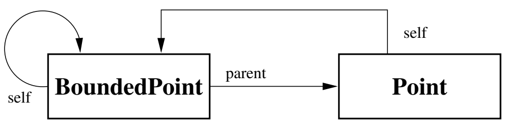
8.1.5 Single inheritance

As a first approach for modelling inheritance, we use *delegation semantics* (as in Self [US87] and Sina [Aks89]): each object has an instance of its parent-class. Therefore, only the exported features of a parent-class can be accessed. This has the consequence that dynamic binding of methods requires a more sophisticated mechanism and is not possible by default. If the subclass redefines a method which is called by another method defined in an ancestor class (and not redefined in the subclass), the original and not the redefined method will be called. As an example, consider the class BoundedPoint which redefines the method move, which in turn gets called by the method double.

The reason why dynamic binding is not supported by this approach is that self within the instance of the parent-class refers to the parent-class object, but not to the subclass object:



To achieve the effect of dynamic binding, self of the parent-class object has to refer to the subclass object:



What we need is an instance of the parent-class where self refers to the subclass object. To do so, we introduce so-called *intermediate objects* where all methods and instance variables of a class are defined, but where `self` is unbound: all methods require an additional binding for `self`.[4] The class metaobject of each class defines an agent `CreateIntermediate` (comparable with a generator in [Coo89]) which acts as a factory for intermediate objects of the class (refer to Figure 8.4).

In the method `Create` of the class metaobject, a new intermediate object is created, each exported method is bound to a method defined in the intermediate object, and the correct binding of `self` is established. Like in the previous model, an empty reference cell is used to model self-references.

Note that besides the implementation of all public class methods and the method `Create`, the class metaobject also exports the method `CreateIntermediate`.

The modelling of inheritance is now straightforward. In order to reuse the methods defined in an ancestor class, the class metaobject of a class gets a fresh copy of the intermediate object of its direct parent-class. This intermediate object is then used to define the intermediate object of the class itself. It is possible to i) override methods, ii) define new methods, and iii) call inherited methods. However, the parent-class has to be explicitly referred to in the method `CreateIntermediate` of a subclass (refer to Figure 8.5). The reader should note that we only use *binding extension* to define the intermediate object of a (sub)class.

Private instance attributes (variables and methods) do not define a binding in the intermediate object of a class; they are only visible within the scope of the corresponding agent `CreateIntermediate`. The same applies for private class attributes (such as the class variable Counter), which are not declared in the interface of a class and, therefore, only visible within the scope of the corresponding class metaobject. As a consequence, unlike in other object-oriented programming languages (e.g. Eiffel [Mey92]), private attributes cannot be accessed by subclasses. This implies that a subclass can define a new (private or public) attribute with the same name without interfering with a private attribute of one of its parent-classes.

In order to define the arguments for all methods of `PointClass`, we have used so-called *nested forms* [Lum99]. Due to the fact that the labels of a form can only bind names (and not forms), is is not possible to communicate more than one form without losing the original structure (e.g. if two forms `F` anf `G` define a binding for a label `foo`, then `<F,G>` "loses" the binding for `foo` in `F`). In order to solve this problem, PICCOLA(F) allows a user to communicate nested forms and translates an expression of the form

```
<label=cname, form=AFormValue>
```

into

```
<label=cname, function form() = AFormValue>
```

This construct also explains why accessing nested forms requires an additional function call.

[4]Using the terminology of Abadi and Cardelli, we call methods with an additional `self` parameter *pre-methods* [AC96].

```
value PointClass =
  let
    value Counter = ref(<val = 0>)              {- class variable -}

    function CreateIntermediate(Init) =
        let
          value x = ref(<val = Init.x>)         {- instance variables -}
          value y = ref(<val = Init.y>)
        in
          Counter.set(<add (<lval=Counter.get().val, rval=1>)>);
          <
            function X(Args) = x.get(),
            function Y(Args) = y.get(),
            function move(Args) =
                x.set(<add(<lval=x.get().val,rval=Args.dx>)>);
                y.set(<add(<lval=y.get().val,rval=Args.dy>)>),
            function double(Args) =
                Args.self().move(<dx=x.get().val,dy=y.get().val>)
          >
        end

    function Create(Init) =
      let
        value PointIntermed = CreateIntermediate(Init)
        value Self = emptyRef()

        value NewInstance =
          <
            function X()=PointIntermed.X(<self=Self.get()>),
            function Y()=PointIntermed.Y(<self=Self.get()>),
            function move(Args)=PointIntermed.move(<Args,self=Self.get()>),
            function double()=PointIntermed.double(<self=Self.get()>)
          >
      in
        Self.set(<NewInstance>);                {- self binding -}
        NewInstance
      end
  in
    <
      Create = Create,
      CreateIntermediate = CreateIntermediate,
      function NoOfPoints() = Counter.get()
    >
  end
```

Figure 8.4: Source code of `PointClass` class metaobject with intermediate objects.

```
value BoundedPointClass =
  let
    function CreateIntermediate(Init) =
        let
          value ParentIntermed = PointClass.CreateIntermediate(<Init>)
          value b = ref(<val = Init.bound>)      {- constant bound -}
        in
          <
            ParentIntermed,                      {- binding extension -}
            function bound(Args) = b.get(),
            function move(Args) =                {- method overriding -}
              let
                value res = add(<lval=Args.self().Y().val,
                                 rval=Args.dy>)
                value b = Args.self().bound()
              in
                if cmp(<lval=res.val,rval=b.val>).val=-1 then
                  ParentIntermed.move(<Args>)
                else <> end
              end
          >
        end

    function Create(Init) =
      let
        value BPInter = CreateIntermediate(Init)
        value Self = emptyRef()
        value NewInstance =
          <
            function X() = BPInter.X (<self=Self.get()>),
            function Y() = BPInter.Y (<self=Self.get()>),
            function move(Args) = BPInter.move (<Args, self=Self.get()>),
            function double() = BPInter.double (<self=Self.get()>),
            function bound() = BPInter.bound (<self=Self.get()>)
          >
      in
        Self.set(<NewInstance>);                 {- self binding -}
        NewInstance
      end
  in
    <
      PointClass,               {- extend parent-class class object -}
      Create = Create,
      CreateIntermediate = CreateIntermediate
    >
  end
```

Figure 8.5: Source code of `BoundedPointClass`.

8.1.6 Observations

Summarizing the basic object model in the FORM calculus, we see that a (base-level) object consists of three parts: an *intermediate object*, an *interface adaptor*, and a set of *request and reply channels*. All request and reply channels represent the service interface of an object. In order to call a particular method, a message representing the actual input parameters has to be sent along the corresponding request channel. The result of this method invocation can be obtained by reading the appropriate reply channel (which is often implicit). In Figure 8.6, request channels are represented by a small square inside and reply channels by a square outside an object, respectively. The solid lines with arrowheads denote channels used for one-way communication within an object and between an object and its class metaobject, and are not accessible by external clients. The intermediate object consists of a set of local channels representing instance variables and a number of agents representing all method implementations. The interface adaptor (corresponding to the method `Create`) maps each request and reply channel to the agent abstraction representing the corresponding method implementation. One may note that there is no limit on the number of concurrently executing methods in an object and that methods are not explicitly synchronized.

In order to reuse (inherit) method implementations, but having a sound interpretation of self-references, intermediate objects still have an unbound self-reference. The actual binding of `self` is achieved in the interface adaptor by passing the value of `self` as a parameter to each method invocation.

A commonly used mechanism in various object-oriented programming languages is to represent classes as objects at a meta-level [KdRB91]. We have used this technique not only to model class features as features of a metaobject, but also to obtain a disciplined way to create objects and to model inheritance. In order to create an object, the class metaobject instantiates an intermediate object, defines a new set of request and reply channels, and binds them all using an appropriate interface adaptor. Inheritance can be achieved by combining and extending intermediate object templates of already existing classes.

The reader should note that the basic object model of Pierce and Turner as well as our extensions belongs to the category of so-called *imperative object models*. In models of this category, object updates are imperative, which implies that self can be bound when an object is instantiated. This is often referred to as *early self binding*. The main advantage of this approach is that the fixed-point operator has to be applied only once, at the time of object instantiation. Self always refers to the same set of local agents and channels, and the state of an object is always modified imperatively by the object's methods.

In an object model that supports functional updates, however, early binding of self does not work, as each change in the state of an object implies the creation of a new object [AC96]. In such a situation, self has to be bound each time a method is invoked, which is often referred to as *late self binding*.

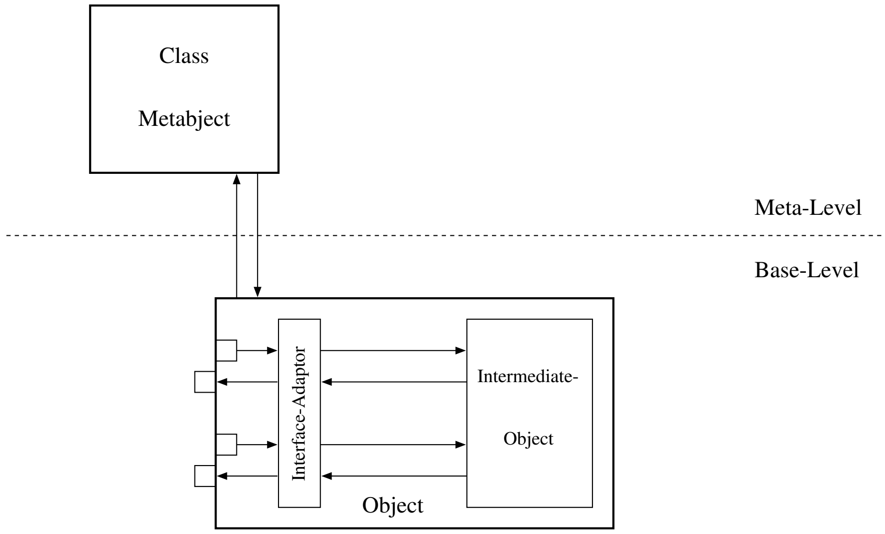


Figure 8.6: Basic object model in the FORM calculus.

8.2 Synchronization

In the previous section, we have discussed a first extension of the Pierce/Turner basic object model, but we mainly concentrated on modelling features of object-oriented programming languages that do not necessarily address concurrency. It is obvious, however, that the presence of concurrent activities within an object requires some degree of synchronization. As a first extension to our basic object model, we will investigate abstractions for synchronizing concurrent objects.

Several synchronization schemes have been proposed to address various levels of concurrency control [Bri96]. Centralized schemes, such as *path expressions*, specify in an abstract way the possible interleavings of method invocations [Ame87, VdBL89]. Decentralized schemes, such as *guards*, are based on boolean activation conditions that may be associated to each method [DLDR$^+$91]. Higher level formalisms are based on the notion of *abstract behaviours* [TV89]. Recent work has tried to integrate these synchronization schemes into a framework for classifying, comparing, customizing, and combining synchronization abstractions for object-oriented concurrent programming [Bri96].

An evaluation of several synchronization schemes and mechanisms revealed that objects are most easily synchronized at a meta-level. This implies that synchronization policies have to be reified as first-class entities. Our experiments showed that McHale's concept of "Generic Synchronization Policies" (GSPs) [McH94] could be easily integrated into the meta-level of our basic object model and that it formed a promising basis for the definition of higher-level, reusable synchronization abstractions. GSPs do not only allow for a complete separation of computational and synchronization abstractions, but enhance the reuse of existing (sequential) components in a concurrent environment. The extension of our object model also allows us to formalize the concept of GSPs.

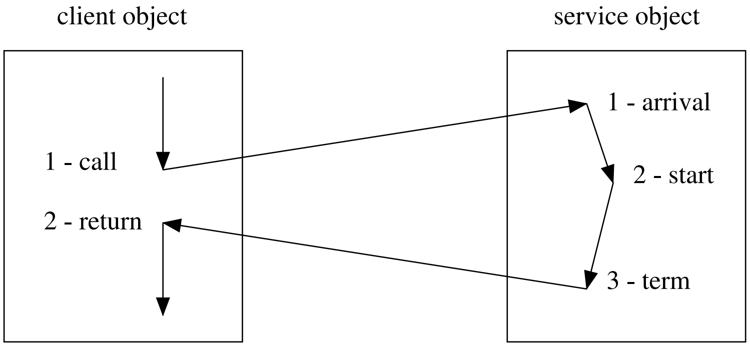


Figure 8.7: The events in the lifespan of a typical method invocation.

8.2.1 Generic Synchronization Policies

The concept of Generic Synchronization Policies provides a mechanism to synchronize objects at the granularity of method invocations and is based on a paradigm called "Service-object Synchronization" (SoS) [McH94]. The paradigm consists of the following four concepts: i) events (and code executed at them), ii) delaying and starting method invocations, iii) accessing information about method invocations, and iv) a strict separation between synchronization code/data and code/data of the object itself.

In order to show how this paradigm works, we first need to describe the sequence of events that takes place when an object invokes an operation upon another object.

From the service object's perspective (which is the only one we will consider), there are three events of interest: *arrival*, *start* and *term* (short for termination) of a method invocation. When an invocation arrives, it may be delayed due to some synchronization constraints. Some time later, it will start execution and, finally, it will terminate execution. We assume that events do not overlap. For example, if two invocations arrive at the same time, we assume that their arrival events will be ordered. The sequence of events is summarized in Figure 8.7.

GSPs permit an action (user code) to be associated with each possible event. The execution of an action will always complete before another event can occur. Synchronization constraints between methods are expressed using the concept of a guards (i.e. a boolean expression). Each invocation will be delayed until the corresponding guard evaluates to true.

In GSPs, the genericity lies in the fact that actions and guards are not associated directly with a particular method of a given object. Conceptually, methods are grouped into *categories* for which actions and guards are specified. At instantiation time, all methods will "inherit" all actions and guards according to their associated category.

The second concept of the SoS paradigm is needed in order to delay a method invocation due to some synchronization constraints and start its execution when all synchronization constraints are fulfilled. The SoS paradigm does not specify how the mechanism for delaying and starting invocation has to be implemented.

In order to express complex synchronization schemes, it is necessary to access information about the method invocations upon an object. In code for actions and guards, the following information needs to be made available:

- the arrival time of the current invocation (for which the action or guard is executed),
- the number of waiting invocations from a given category,
- the number of executing invocations from a given category,
- a list of all waiting invocations from a given category, and
- the method's parameters.

Other information could be added, like, for example, the number of terminated invocations or the list of all executing invocations from a given category. In our modelling, however, we restricted ourselves to the information mentioned above.

Finally, the SoS paradigm requires a strict separation between synchronization code/data and code/data of the object itself. As an example, synchronization code (guards and actions) cannot access instance variables of the object and vice versa, ensuring that concurrent execution of synchronization and sequential code cannot interfere (e.g. synchronization code cannot access information currently being updated by sequential code). This requirement also ensures that it is not only possible to completely separate the specification and implementation of synchronization code from sequential code, but also allows us to reuse synchronization policies in different settings.

As an example, we present the (well-known) Readers-Writers policy. The policy has two categories of methods: one category representing methods that only read instance variables, and another category representing methods that change the value of some instance variables of an object. Using GSPs, the Readers-Writers policy could be specified as follows (we use the same syntax as in [McH94]):

```
policy ReadersWriters [ReadOps, WriteOps] {
      function ReaderAllowed (t: Invocation): Bool
      begin
            return exec(WriteOps) = 0;
      end
      function WriterAllowed (t: Invocation): Bool
      begin
            return exec(ReadOps)+exec(WriteOps) = 0;
      end
      map  guard(ReadOps) → ReaderAllowed
           guard(WriteOps) → WriterAllowed
}
```

This policy specifies that a Read operation can only take place when there is no executing Write operation (i.e. exec(WriteOps) = 0) and a Write operation can only take place when no other operation is executing (exec(ReadOps)+exec(WriteOps) = 0). The construct

```
map  guard(ReadOps) → ReaderAllowed
     guard(WriteOps) → WriterAllowed
```

binds the guards for the different method categories.

The Readers-Writers synchronization policy defined above can be used in a class-based way to synchronize objects where all methods are either reader or writer methods, like in the example below:

```
class Point {
    . . .          /* defines X, Y, move, and double */
    synchronization
        ReadersWriters [ {X Y}, {move double} ]
}
```

8.2.2 Modelling GSPs in the FORM calculus

In the following, we illustrate the integration of GSPs into the FORM calculus based object model. First, this shows that the basic object model is open enough to be extended with additional features like GSPs, and second it demonstrates that the GSP paradigm can be easily adapted to any object model.

In order to model GSPs, we use a similar architecture as for modelling normal objects: synchronization policies are represented as objects in the meta-level (and therefore they behave as metaobjects) while so-called *method wrapper objects* providing an interface for synchronization are part of the base-level. The main difference of modelling GSPs compared with our modelling of plain objects is that the policy metaobjects are not linked to method wrapper objects a priori: this binding is only established when an object is instantiated. Furthermore, method wrapper objects are fully generic: they can wrap any method and can be bound to any policy. Finally, the binding to a policy can be changed at run-time. The overall structure of the GSP integration is shown in Figure 8.8.

Synchronization wrappers. Like McHale, we have also opted for the technique of placing a synchronization wrapper around the code to be synchronized. This technique is commonly employed in the implementation of synchronization mechanisms. The main idea of the synchronization wrappers is to take a pure unsynchronized object and to wrap its methods in a pre- and post-synchronization code. For example a method like

```
method foo()
begin
    body;
end
```

will be transformed (wrapped) into

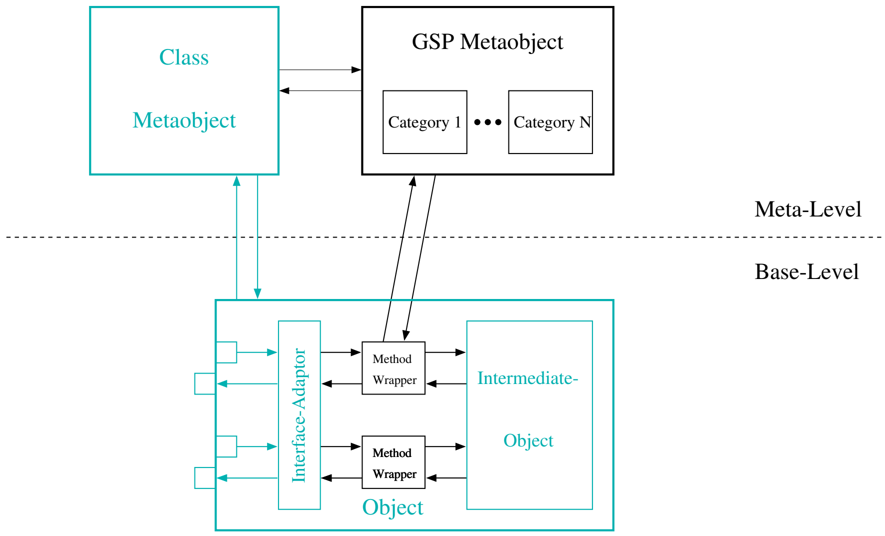


Figure 8.8: Integration of GSPs in the object model.

method foo()
begin
    pre-synchronization code;
    body;
    post-synchronization code;
end

In the following, we assume that a method is represented as the agent

$$\textbf{def}\ M(X)\ =\ A$$

and that X is a free variable in the agent A, which returns the result of a method invocation (a value or just a signal of termination) using the channel X_{result}. Wrapping a method M leads to the following agent:

$$\begin{aligned}\textbf{def}\ WM(Y)\ =\ (\nu\, a_w, r_w, r_p, r_o)(\ & PreWrapper \mid r_p(Z).\overline{a_w}(Z\langle result = r_w\rangle) \\ & \mid a_w(X).A \\ & \mid r_w(Z).(PostWrapper \mid r_o(R).\overline{Y_{result}}(R)\ )\ )\end{aligned}$$

The wrapping process itself is modelled through an object which has at least two methods. One method is used to execute the synchronized method (compare with the given wrapped agent WM) and the other method is used to set the policy specific method wrappers. The following agent illustrates the wrapping:

$$\begin{aligned}&(\nu\ PreWrapper,\ PostWrapper)\\ &\qquad (\ SetWrapper(X).(\ PreWrapper(\_).\overline{PreWrapper}(X)\\ &\qquad\qquad\qquad \mid PostWrapper(\_).\overline{PostWrapper}(X)\\ &\qquad \mid\ WM\ )\end{aligned}$$

The reader should note that the formal parameter X of the $SetWrapper$ agent requires a binding for the labels pre and $post$, which denote the access points for the pre- and post-wrappers, respectively.

The wrapper objects behave as two-way generic communication interceptors. The reason why we had to introduce such objects is that PICCOLA(F) does not support message interception. Such a facility could be added to the language through an extension of the semantics of channel creation. The creation of a new channel would be mapped internally to an agent which can be parameterized with an input and output related agent which is activated when an input or output takes place. After the interceptor agent has finished, the initiated communication proceeds as usual.

Binding a GSP to an object. The Readers-Writers policy described in section 8.2.1 can be represented by the following agent:

$$\begin{aligned}\mathbf{def}\ \mathit{ReadersWritersPolicy}(X) \ = \ & (\ \mathit{ReaderAllowed}(Invocation) \\ & \mid \mathit{WriterAllowed}(Invocation) \\ & \mid Map(\langle cat = X_{readers}, guard = \mathit{ReaderAllowed}\rangle) \\ & \mid Map\langle cat = X_{writers}, guard = \ \mathit{WriterAllowed}\rangle))\end{aligned}$$

This agent takes as arguments the methods belonging to the appropriate categories. Then the agent $\mathit{ReadersWritersPolicy}$ starts four agents in parallel: $\mathit{ReaderAllowed}$ and $\mathit{WriterAllowed}$ which are the policy specific synchronization guards and two agents Map which immediately end after binding the methods of a category to its synchronization guards.

An initial binding of a GSP is done at object creation time. In order to add GSPs to the basic object model, the method $Create$ of the class metaobject has to be modified: first, $Create$ takes an additional parameter $Policy$ which defines a binding for the actual synchronization policy, and second, two additional agents ($\mathit{CreateWrapped}$ and $BindPolicy$) have to be added to the body of $Create$. The following agent illustrates the modified method:

$$\begin{aligned}\mathbf{def}\ \mathit{Create}(X) \ = \ & (\nu\ Intermediate, \mathit{WrappedIntermediate}, \mathit{NewInstance}, \mathit{Self}) \\ & (\ \mathit{CreateIntermediate}(Intermediate) \\ & \mid \mathit{CreateWrapped}(\langle int = \ Intermediate, wrap = \ \mathit{WrappedIntermediate}\rangle) \\ & \mid CreateInstance(\langle wrap = \ \mathit{WrappedIntermediate}, inst = \ \mathit{NewInstance}\rangle) \\ & \mid BindPolicy(\langle inst = \ \mathit{NewInstance}, pol = X_{policy}, \mathit{self} = \ \mathit{Self}\rangle) \\ & \mid \mathit{Self}(Y).\overline{X_{result}}(Y)\ )\end{aligned}$$

In order to create a new object, the method $Create$ receives a synchronization policy in X_{policy} and a reply channel in X_{result} (which is used to return the new created object to the caller). The method $Create$ starts four agents in parallel which are synchronized through its communicated channels: i) $\mathit{CreateIntermediate}$ sends an intermediate object along the channel $Intermediate$, ii) $\mathit{CreateWrapped}$ takes this object and sends an intermediate object with empty synchronization wrappers along $\mathit{WrappedIntermediate}$, iii) $CreateInstance$ creates the interface adapter and sends the resulting object along

$NewInstance$, and iv) $BindPolicy$ takes this instance, binds it to the given policy, and establishes the correct binding of $self$.

With this implementation, we have an *object-based* approach of the GSP mechanism because the object creation is parameterized with the actual synchronization policy. McHale proposed that the synchronization code be class based and the synchronization policy is part of the class definition. We use an object based approach in order to allow that different synchronization policies can be assigned to objects of the same class, and that policies can be changed at run-time.

As mentioned above, a policy is represented as an object (see Figure 8.8). The policy object is a metaobject for the method wrapper objects as well as for the synchronized object. The method wrapper objects themselves do not have any instance variables. All method wrapper objects use exclusively the synchronization variables provided by the policy metaobject. Therefore, the synchronization variables can been seen as class variables and the policy methods as class methods, respectively.

8.2.3 Observations

Our experiments have shown that the concept of generic synchronization policies is a promising basis for the definition of higher-level, reusable synchronization abstractions. The integration of GSPs illustrates that the basic object model is open enough to be easily extended with additional abstractions, and that the use of meta-level abstractions allows for an easy and flexible integration of additional abstractions without the need to change the underlying model. It is important to note that the binding of a GSP only affects the creation of an object: the external interface of an object remains the same. The original specification described in [McH94], however, has a few drawbacks, which will be discussed in the following.

First of all, GSPs are used to synchronize concurrent *objects*, but their instantiation is restricted to *classes*: each instance of a class has the same synchronization code. Due to the fact that the concept of GSPs is independent of classes, we have extended it in our modelling in order to decide at object creation time which synchronization policy is bound to an object. Therefore, it is possible that not all instances of a class are synchronized using the same synchronization policy.

McHale claims that both external and self calls of methods should be synchronized by the same synchronization mechanism [McH94], but we argue that there are good reasons why a single layer/scheme of synchronization is not enough. Both methods move and double of the class Point modify the state of a Point instance and, therefore, have to be considered as writer methods. Hence, in a Readers-Writers policy, only one of these two methods can be active at a time. However, the body of the method double calls move in order to update the coordinates of a Point instance, which implies that two writers methods have to be active (i.e. the method double invoked by a client and the method move called by double). This is not possible in a Readers-Writers policy, and a call of double will cause a deadlock situation. In such a situation, McHale proposes to rewrite the corresponding class in a way that a writer method never calls another writer method by introducing unsynchronized private methods (e.g. the class Point defines a method doMove which updates the coordinates and is called by both move and double). However, from our point of view, changing the implementation of a class is not an acceptable solution, especially in the context of black-box composition. We argue that self calls of methods should be synchronized differently, and in order to guarantee consistency of an objects internal state, there is a need for a second layer of synchronization. This extension is a topic for future research.

Similar to the problem of self calls is the fact that the concept of GSPs does not take care of method invocations that recursively call other methods of the same object by external clients (e.g. method foo of object A calls a method of object B, which calls method bar of A). There is no possibility to assign an information to the invocation of bar that this invocation is a direct cause of the execution of foo, and that it should be synchronized differently.

Finally, McHale claims that a hybrid concept[5] can be implemented using two variables (a synchronization and a instance variable), and that there are only *hypothetical* cases where this is not possible. We think that such cases are far form being hypothetical: consider a bounded container object (e.g. a bounded list) which does not store any duplicates. In such a situation, it is not trivial to consistently update both variables containing the information about the number of items stored. McHale proposes a work-around for such situations [McH94], which we think either needs to be revised or the underlying concept has to be modified.

8.3 Class abstractions

In the previous two sections, we have defined a basic object model in the FORM calculus (section 8.1) and showed how this model can be extended with synchronization abstractions (section 8.2). As a next step towards a common metamodel for object-oriented abstractions, we introduce *class abstractions* which allow us to define class metaobjects in a more natural way (i.e. by using appropriate parameterizations of class abstractions). We also show that *polymorphic form extension* is an essential feature for modelling class abstractions in general and for different flavours of inheritance and method dispatch in particular.

8.3.1 From pre-methods to generators

Analyzing the class metaobjects `PointClass` and `BoundedPointClass` shown in Figure 8.4 and 8.5, it becomes clear that the class method `CreateIntermediate` defines the *behaviour* of its instances whereas `Create` is used to i) instantiate an intermediate object and ii) to establish a correct binding of `self`. What prevents us from defining a generic class method `Create` is the fact that each exported method has to be

[5] A hybrid concept denotes information that is both needed for synchronization and computation (e.g. the number of items in a bounded buffer).

bound to a method defined in the intermediate object by passing an additional parameter for `self` (i.e. self-binding for pre-methods).

Cook and Palsberg [CP94] have proposed an approach for modelling classes, mixins, inheritance etc. using the notion of *generators* and *wrappers*. A generator, denoted by G, defines the behaviour of objects (encoded as records) with an unbound self-reference whereas a wrapper, denoted by W, establishes the correct binding of self. In fact, a generator is a *lambda abstraction* over self (i.e. a function that requires a parameter for self) and a wrapper is the *fixed-point operator* for the corresponding generator. A similar approach has also been proposed by Reddy, but the explicit separation between generators and wrappers is omitted [Red88].

The notion of generators and wrappers can be easily adapted to our basic object model and allows us to simplify the encoding. We omit the additional parameter `self` for all methods and pass the self-reference as a parameter to the method `CreateIntermediate` of the corresponding class metaobject (i.e. pre-methods are replaced by *generator methods* [CP94]). In order to model the fixed-point operator, we use a different approach: due to the fact that the correct value of self is not known when the method `CreateIntermediate` is invoked, we pass a *function* `self` which returns the correct binding for self when it is called (i.e. it returns the contents of the reference cell `Self` defined in the body of `Create`). This allows us to establish the binding of self like in the previous model. As a consequence of this adaptation, the explicit binding of exported methods to methods defined in the intermediate object can be omitted; we simply return the result of the corresponding `CreateIntermediate` call. Therefore, the code for the class method `Create` is completely generic and can be used for any class metaobject.

As an additional extension to the previous models, we introduce a self-reference for class metaobjects as well. The reader should note that a self-reference for class metaobjects is not needed in this stage of modelling classes. However, we will need this feature further on in this section in order to enhance flexibility and in particular extensibility.

Refer to Figure 8.9 for the code of the modified class metaobject `PointClass`. The code for `BoundedPointClass` is very similar (only the class method `CreateIntermediate` has to be adapted accordingly) and has therefore been omitted here.

8.3.2 Smalltalk-like behaviour

In the previous section, we have adapted the basic object model in a way that the class method `Create` is fully generic and can be used for any class metaobject. In this section, we go a step further and define a generic class method `CreateIntermediate` as well. As a result, we illustrate the specification of a class abstraction which defines the desired behaviour of the class metaobjects previously used (i.e. dynamic binding of self-calls, single inheritance etc.).

Inheritance in many object-oriented programming languages is a mechanism of *incremental derivation* of classes. In the previous examples, we have directly modified the behaviour of a parent-class in order to achieve the required behaviour of a class. From a different point of view, a subclass can be viewed as an abstraction which specifies how it

```
value PointClass =
  let
    value MetaSelf = emptyRef()     {- self reference of class object -}
    function Mself() = MetaSelf.get()
    value Counter = ref(<val = 0>)              {- class variable -}

    function CreateIntermediate(Init) =
        let
          function self() = Init.self()        {- helper function -}
          value x = ref(<val = Init.x>)        {- instance variables -}
          value y = ref(<val = Init.y>)
        in
          Counter.set(<add (<lval=Counter.get().val, rval=1>)>);
          <
            function X() = x.get(),
            function Y() = y.get(),
            function move(Args) =
                x.set(<add(<lval=x.get().val,rval=Args.dx>)>);
                y.set(<add(<lval=y.get().val,rval=Args.dy>)>),
            function double() =
                self().move(<dx=x.get().val,dy=y.get().val>)
          >
        end

    function Create(Init) =
      let
        value Self = emptyRef()     {- self reference of instances -}
        function self() = Self.get()
        value NewInstance = Mself().CreateIntermediate(<Init,self=self>)
      in
        Self.set(<NewInstance>);              {- self binding -}
        NewInstance
      end

    value ClassInstance =
      <
        Create = Create,
        CreateIntermediate = CreateIntermediate,
        function NoOfPoints() = Counter.get()
      >
  in
    MetaSelf.set (<ClassInstance>);
    ClassInstance
  end
```

Figure 8.9: Replacing pre-methods with generator-methods in `PointClass`.

differs from its parent-class. This difference may be indicated as a set of changes or as a *delta*. In the case of the class BoundedPoint, the set of changes contains the private instance variable b, the additional method bound, and the modified method move.

In order to formalize the process of inheritance, we use the notion of generators and wrappers introduced in the previous section. Using the approach of Bracha and Cook [BC90], we define the generator for a class as the composition of its parent-class generator (or a collection of parent-class generators in case of multiple derivation) and an *incremental modification*, written Δ.

Throughout the rest of this chapter, we will use the following notations that facilitate the presentation of our formalism. These notations can be though of as syntactic sugar on top of the FORM calculus: there exists a sound interpretation of them in terms of the primitives of the calculus. The expression $\{m_1(X_1) \rightarrow b_1, \ldots, m_n(X_n) \rightarrow b_n\}$ denotes a form with fields $m_1, ..., m_n$ representing methods with the method bodies $b_1, ..., b_n$ and the (keyword-based) formal arguments $X_1, ..., X_n$, respectively. In fact, this expression stands for an interface of an active object (i.e. it is represented by a FORM calculus agent). The projection of a label l of a form value F is denoted by $F.l$. The update of a field l with value v is denoted by $\{l = v\}$. We use the operators $\oplus$ and $\setminus$ to denote the polymorphic extension and polymorphic restriction of two forms, respectively. Finally, we use "let $V = e_1$ in e_2" to assign expression e_1 to V in e_2 and $\mathit{fix}_X[f(X)]$ to denote the least fixed-point of a function f.

Using Δ_{Point} to stand for the incremental modification defined by class Point, the generator G_{Point}, the wrapper W_{Point}, and I denoting the constructor arguments, the class Point is defined as follows:[6]

$$
\begin{array}{rcl}
\Delta_{Point}(I) & = & \text{let } x = I.x,\ y = I.y \\
& & \text{in} \\
& & \quad \{\ X() \rightarrow x,\ Y() \rightarrow y, \\
& & \quad\ \ \mathit{move}(\mathit{Args}) \rightarrow x = x + \mathit{Args}.dx;\ y = y + \mathit{Args}.dy, \\
& & \quad\ \ \mathit{double}() \rightarrow I.\mathit{self}.\mathit{move}(\{dx = x, dy = y\})\ \} \\
G_{Point}(I) & = & \Delta_{Point}(I) \\
W_{Point}(I) & = & \mathit{fix}_s\,[\,G_{Point}\ (I \oplus \{\mathit{self} = s\})\,] \\
\mathit{Point} & = & \{W(I) \rightarrow W_{Point}(I),\ G(I) \rightarrow G_{Point}(I)\}
\end{array}
$$

The expression $\mathit{fix}_s\,[\,G_{Point}(I \oplus \{\mathit{self} = s\})]$ yields an object with an appropriately bound self-reference, which is expressed by the binding $\mathit{self} = s$.

The class BoundedPoint, the corresponding incremental modification Δ_{BPoint}, generator G_{BPoint}, and wrapper W_{BPoint} are defined as follows:

$$
\begin{array}{rcl}
\Delta_{BPoint}(I) & = & \text{let } b = I.b \\
& & \text{in} \\
& & \quad \{\ \mathit{bound}() \rightarrow b, \\
& & \quad\ \ \mathit{move}(A) \rightarrow \mathit{if}\ (I.\mathit{self}.Y() + A.dy) < I.\mathit{self}.\mathit{bound}() \\
& & \qquad\qquad\qquad\qquad \mathit{then}\ I.\mathit{orig}.\mathit{move}(A)\ \}
\end{array}
$$

[6] In order to enhance readability, we have omitted the class variable Counter and method NoOfPoints.

$$\begin{array}{rcl} G_{BPoint}(I) & = & \text{let } Obj_I = Point.G(I) \\ & & \text{in} \\ & & \quad Obj_I \;\oplus\; \Delta_{BPoint}\ (I \oplus \{orig = Obj_I\}) \\ W_{BPoint}(I) & = & fix_s\, [\, G_{BPoint}\ (I \oplus \{self = s\})\,] \\ BoundedPoint & = & \{W(I) \rightarrow W_{BPoint}(I),\ G(I) \rightarrow G_{BPoint}(I)\} \end{array}$$

In order to correctly dispatch self-calls and to have a sound interpretation of the inherited parent behaviour in *BoundedPoint*, the abstraction Δ_{BPoint} requires two parameters: $self$ (the argument I contains a binding for $self$) and $orig$ that gives access to the inherited behaviour. Note that i) the generator G_{BPoint} defines a composition of the intermediate object Obj_I (instantiated by the generator G_{Point}) and the intermediate object generated by Δ_{BPoint} and ii) the application of the fixed-point operator in W_{BPoint} on G_{BPoint} (i.e. the composition of both intermediate objects) ensures a correct binding of $self$ within the resulting object [LSN96].

Analyzing the examples, we can see that the two wrappers W_{Point} and W_{BPoint} are almost identical: they only differ in the used generator. Furthermore, the composition of the two intermediate objects in G_{BPoint} is very similar to the way inheritance is defined in Smalltalk [GR89], and it can be assumed that the underlying inheritance mechanism does not differ in the same semantic model. Hence, by appropriately parameterizing the corresponding generators and wrappers, the same G and W abstractions can be used for different classes.

This observation motivates the definition of the class abstraction $Class$ which specifies a Smalltalk-like class model as follows (we use $\lambda X \rightarrow b$ to denote an anonymous function with parameter X and function body b):

$$\begin{array}{l} Class(A) \;=\; fix_{Mself}\ [\ \lambda Mself \rightarrow \\ \qquad \{\ G(I) \ \rightarrow\ \text{let } Obj_I = A.P.G(I) \text{ in } Obj_I \oplus A.\Delta(I \oplus \{orig = Obj_I\}), \\ \qquad\ \ W(I) \ \rightarrow\ fix_s\ [\ Mself.G(\{init = I, self = s\})\ ]\ \}\ \oplus A\ ] \end{array}$$

$Class$ is a *function* which expects a keyword-based argument A with appropriate bindings for Δ and P denoting the incremental modification and the parent-class for a class to be defined, respectively. Evaluating this function yields a class metaobject with self-reference $Mself$.

The implementation of the class abstraction `Class` shown in Figure 8.10 corresponds to the abstraction *Class* defined above: `CreateIntermediate` implements the generator behaviour G whereas `Create` corresponds to the wrapper W. Although the class method `CreateIntermediate` is exported under its required name, it is internally defined as `SmallIntermediate`. This is a naming convention for the fact that the abstraction `Class` defines a Smalltalk-like class model (i.e. dynamic binding of self-calls, direct invocation of inherited methods, single inheritance etc.) and that it is possible to distinguish it from other generators with different behaviour.

Note that a *closure* is used to define the appropriate behaviour of `SmallIntermediate`: the body refers to both the function `Delta` and the class metaobject `parent` passed as an argument to `Class`. Furthermore, the order of the label bindings for the value `ClassInstance` does not matter if we only consider the abstraction `Class` on its own, but it is important for further extensions and adaptations.

In order to break the recursion in the generators (i.e. the generator of a class C always refers to the generator of its parent-class), it is necessary to introduce a common ancestor for all classes which acts as the root of the class hierarchy and defines appropriate default behaviour.[7] For our modelling purposes, we have chosen a similar approach as is used in Smalltalk: although `Object` is the root of the hierarchy, it is still defined using the abstraction `Class`. It internally defines an incomplete class metaobject `Top` which breaks the the recursion of `CreateIntermediate` calls (refer to Figure 8.11 for details). The method `DeltaInner` of `Object` is used for Beta-style inheritance (see section 8.3.4).

Given a function `Delta` and parent-class `parent`, `Class` creates a class metaobject with the desired behaviour for both instance creation and incremental derivation. Therefore, the class metaobject `PointClass` can be created as follows:

```
value PointClass = Class (
  let
    value Counter = ref(<val = 0>)
  in
    <
      parent = Object,
      function Delta(Init) = ...
             {- same code as CreateIntermediate in Figure 8.9 -}
      function NoOfPoints() = Counter.get()
    >
  end
)
```

The code for `BoundedPointClass` is very similar, and the `Delta` defines i) the private instance variable b and ii) the code for the two methods move and bound. However, in contrast to the code given in Figure 8.5, `Delta` does not make any explicit reference to its parent-class `PointClass`; the parent-class is specified using the parameter `parent` passed to the abstraction `Class`.

It is important to note that the behaviour of the instances of a class is not fully specified by the abstraction Δ and the parent-class. As we will show in the next sections, Δ only defines the *difference* with respect to the parent-class, but does not specify how self-calls are dispatched.

8.3.3 Static and dynamic binding

In the previous section, we have illustrated that a combination of (parameterized) deltas, generators, and fixed-point operators can be used to define a Smalltalk-like class model.

[7] Not all object-oriented programming languages have such a root class as a common ancestor to all classes (e.g. C++). However, it is considered to be good practice to introduce a user-defined root class for developing frameworks and applications in these languages.

```
function Class(Args) =
  let
    value MetaSelf = emptyRef()      {- self reference of class object -}
    function Mself() = MetaSelf.get()        {- helper abstraction -}

    function SmallIntermediate(Pars) ={- definition of inheritance -}
      let
        value ParentIntermed = Args.parent().CreateIntermediate(<Pars>)
      in
        < {- polymorphic form extension -}
          ParentIntermed,
          Args.Delta(<Pars, orig = ParentIntermed>)
        >
      end

    function Create(Init) =                  {- default constructor -}
      let
        value Self = emptyRef()      {- self reference of instances -}
        function self() = Self.get()         {- helper abstraction -}

        value NewInstance =
            Mself().CreateIntermediate(<init=Init,self=self>)
      in
        Self.set(<NewInstance>);             {- binding of self -}
        NewInstance
      end

    value ClassInstance =
      < {- meta abstraction for inheritance -}
        CreateIntermediate = SmallIntermediate,

        {- default constructor -}
        Create = Create

        {- userdefined parameters and methods -}
        Args
      >
  in
    MetaSelf.set(<ClassInstance>);
    ClassInstance
  end
```

Figure 8.10: Class abstraction for a Smalltalk-like class model.

```
value Object = Class (
   let
      value Top = <function CreateIntermediate(Pars) = <>>
   in
      <
         parent = Top,
         function Delta(Init) = <>,
         function DeltaInner() = <>
      >
   end
)
```

Figure 8.11: Class `Object` as a common ancestor of all classes.

Interestingly enough, the same concepts and operators can also be applied to define i) pure static and ii) a mixture of both static and dynamic method dispatch (as it is available in C++ [Str91]). In this section, we discuss how the concepts and operators can be used to achieve this behaviour and present the encoding of the corresponding class abstractions.

In the models presented in section 8.1, static method dispatch could only be achieved by declaring the corresponding method in the scope of the object factory and calling it directly (refer to section 8.1.4 for details). However, calling such a method using a self-call still results in dynamic dispatch. In order to prevent a dynamic dispatch in the context of a subclass, we have to find a different solution.

As a possible solution to achieve the desired behaviour, we simply move the fixed-point operator into the generator:

$$
\begin{aligned}
StaticClass(A) \; = \; & \mathit{fix}_{Mself}\ [\ \lambda Mself \rightarrow \\
& \{\ G(I) \;\rightarrow\; \mathit{fix}_s\ [\ \text{let } Obj_I = A.P.G(I \oplus \{self = s\}) \text{ in} \\
& \qquad\qquad Obj_I \oplus A.\Delta(I \oplus \{orig = Obj_I, self = s\})\ ], \\
& \quad W(I) \;\rightarrow\; Mself.G(\{init = I\})\ ]\ \}\ \oplus A\ ]
\end{aligned}
$$

In contrast to the generator used for the abstraction $Class$, the generator of $StaticClass$ is not an abstraction over $self$, as an object with an already bound self-reference is returned (i.e. $self$ within the object created by the generator always refers to itself, and not to the overall object it is part of). This solution has several advantages:

- self-calls within the object created by the generator always refer to itself, regardless in which (parent-)class the receiver is defined,
- it integrates smoothly into the existing model, and
- the corresponding class abstraction can be defined in PICCOLA(F) as a simple extension of the previously defined abstraction `Class` (refer to Figure 8.13).

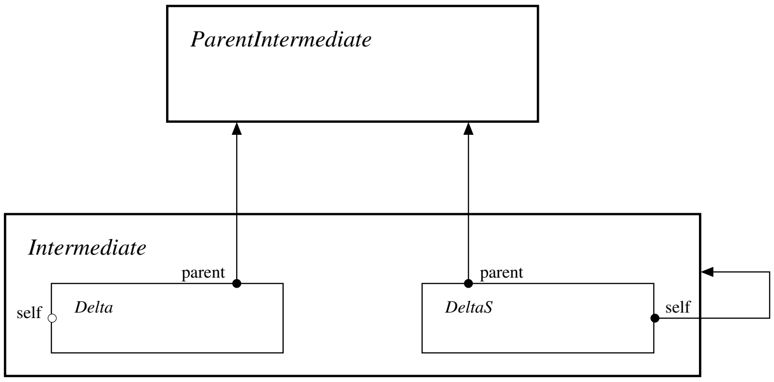


Figure 8.12: Intermediate object created by the generator of $C{+}{+}Class$.

Note that this encoding only ensures static method dispatch for self-calls for the object created by the generator; self-calls made "outside" this object (e.g. an overridden method in a subclass calls another overridden method) are not affected by a static method dispatch.

A class abstraction $C{+}{+}Class$ which allows for a mixture of both static and dynamic method dispatch can be encoded similarly. However, such a class abstraction requires two deltas: one delta for methods to be dynamically dispatched and a second delta for methods with static dispatch. In the following, we denote the former as Δ and the latter as Δ_s, respectively. Again, we apply a fixed-point operator in the generator to achieve static method dispatch:

$$
\begin{array}{rcl}
C{+}{+}Class(A) & = & \mathit{fix}_{Mself}\ [\ \lambda Mself \rightarrow \\
\{\ G(I) & \rightarrow & \mathit{fix}_s\ [\ \text{let } self' = I.self \oplus s,\ Obj_I = A.P.G(I \oplus \{self = self'\}) \text{ in} \\
& & \quad Obj_I \oplus A.\Delta_s(I \oplus \{orig = Obj_I,\ self = self'\}) \oplus \\
& & \quad A.\Delta(I \oplus \{orig = Obj_I,\ self = self'\})\ ], \\
W(I) & \rightarrow & \mathit{fix}_s\ [\ Mself.G(\{init = I,\ self = s\})\ ]\ \}\ \oplus A\ ]
\end{array}
$$

The generator G illustrated above composes the locally defined self-reference (denoted by s) with the self-reference passed as parameter $I.self$ in order to define the correct notion of self (denoted by $self'$) passed to both Δ and Δ_s. The expression $I.self \oplus s$ gives precedence to the methods which are statically bound, and fix_s ensures that in the resulting intermediate object, statically bound methods are dispatched in the desired way. Note that the generator is an abstraction over $self$ and, therefore, behaves correctly in the context of incremental derivation. Figure 8.12 illustrates the behaviour of this generator (*Delta* and *DeltaS* denote the (intermediate) objects created by Δ and Δ_s, respectively; the empty circle on the left-hand side without an attached arrow indicates that `self` for the *Delta* "subobject" is still unbound).

The encoding of the class abstraction $C{+}{+}Class$ is shown in Figure 8.13 (the encoding of $StaticClass$ is similar and has therefore been omitted here). The locally defined reference cell `newSelf` corresponds to the encoding of s whereas the function `self` encodes

```
function CppClass(Args) =
  let
    function CppIntermediate(Pars) =      {- definition of inheritance -}
      let
        value newSelf = emptyRef()        {- local self reference -}
        function self() = <Pars.self(), newSelf.get()>

        value PI =
          Args.parent().CreateIntermediate(<init=Pars.init,self=self>)
        value Stat = Args.DeltaS(<init=Pars.init,self=self,orig=PI>)

        value NewInstance =
          < {- polymorphic form extension -}
            PI,
            Stat,
            Args.Delta(<init=Pars.init, self=self, orig=PI>)
          >
      in
        newSelf.set(<Stat>);              {- bind static self calls -}
        NewInstance
      end
  in
    Class(<Args, CreateIntermediate = CppIntermediate>)
  end
```

Figure 8.13: Class abstraction for both static and dynamic method dispatch.

the composition of $I.self$ and s. Due to the fact that (like in the Smalltalk class model) the wrapper abstraction is not changed, it is possible to define the abstraction `CppClass` as a simple extension of `Class`; only the generator has to be adapted. The expression

```
Class (<Args, CreateIntermediate = CppIntermediate>)
```

creates a class metaobject with a Smalltalk-like behaviour, but since the binding for `CreateIntermediate` is overridden, the required behaviour is achieved (i.e. due to the way the class abstraction `Class` is defined, the overridden `CreateIntermediate` is both exported and called in the body of `Create`).

8.3.4 Beta-style inheritance

Inheritance in the programming language Beta is designed to provide security from replacement of a method by a completely different method [MMPN93]. Beta supports inheritance by *prefixing* of definitions and employs a single definitional construct, the *pattern*, to express types, classes, and methods. We will use the example given below in order to explain the notion of Beta-style inheritance:[8]

[8] A reader familiar with Beta may notice that a slightly simplified syntax is used.

```
Person: class
(# name: string;
   display: virtual proc
            (# do name.display; inner #);
#);

Graduate: class Person
(# degree: string;
   display: extended proc
            (# do degree.display; inner #);
#);
```

The definition of the pattern Graduate is said to be *prefixed* by the pattern Person. In terms of Beta, Person is the *superpattern* of Graduate whereas Graduate is a *subpattern* of Person. The method display is declared to be **virtual** which means that it can be *extended* by a subpattern. However, this does not mean that it can be arbitrarily redefined, as we discuss below.

The behaviour of the method display of the pattern Person is to display the attribute name and then to perform the **inner** statement. For Person, which has no inner behaviour, the **inner** statement is simply a null operation. When a subpattern of Person is defined, the **inner** statement will execute the corresponding method display in the subpattern.

The subpattern Graduate extends the behaviour of the method display of the pattern Person by supplying inner behaviour. For an instance of the pattern Graduate, the initial effect of an invocation of display is the same as for an instance of Person: the original method of Person is executed. After the attribute name is displayed, the inner procedure supplied by Graduate is executed in order to display the attribute degree defined in Graduate. The use of **inner** within Graduate is again interpreted as a null operation; it only has an effect if the method display is extended in a subpattern of Graduate.

In order to formalize the interpretation of a class abstraction $BetaClass$ for a Beta-style class model, we will again use an approach of generators and wrappers:

$$\begin{array}{rcl} BetaClass(A) & = & \mathit{fix}_{\mathit{Mself}}\ [\ \lambda \mathit{Mself} \rightarrow \\ \{\ G(I) & \rightarrow & \text{let } \mathit{Obj}_I = A.\Delta(A.P.\Delta_{\mathit{Inner}}() \oplus A.\Delta_{\mathit{Inner}}() \oplus I) \text{ in} \\ & & \quad \mathit{Obj}_I \oplus A.P.G\ (I \oplus \mathit{Obj}_I)\}\ ), \\ W(I) & \rightarrow & \mathit{fix}_s\ [\ \mathit{Mself}.G(\{\mathit{init} = I, \mathit{self} = s\})\ ]\ \}\ \oplus A\ ] \end{array}$$

The generator of the abstraction $BetaClass$ asymmetrically composes the modifications specified by Δ with the behaviour specified by the superpattern (denoted by $A.P.G(I \oplus \mathit{Obj}_I)$ above). This ensures that the prefix methods have precedence over the suffix methods. Furthermore, the generator requires an additional abstraction Δ_{Inner} (specifying a set of *null methods* for all methods defined in Δ [BC90]), which is passed to the abstraction $BetaClass$ together with Δ and P. Note that this Beta-specific extension substantially benefits from the keyword-based argument passing of the underlying FORM

```
function BetaClass(Args) =
  let
    value MetaSelf = emptyRef()
    function Mself() = MetaSelf.get()

    function BetaIntermediate(Pars) =      {- definition of inheritance -}
      let
        value Inner = <Args.parent().DeltaInner(), Args.DeltaInner()>
        value SelfDelta = Args.Delta (<Inner, Pars>)
      in
        < {- polymorphic form extension -}
          SelfDelta,
          Args.parent().CreateIntermediate(<Pars, SelfDelta>)
        >
      end
  in
    Class (<Args, CreateIntermediate = BetaIntermediate>)
  end
```

Figure 8.14: Class abstraction for a Beta-style class model.

calculus. Using the composition of two objects with null methods in the generator (i.e. the subexpression $A.P.\Delta_{Inner}() \oplus A.\Delta_{Inner}()$) guarantees the correct execution of an **inner** statement if no subpattern is specified.

The specification of $BetaClass$ does not directly encode the restriction that **inner** within a method can only refer to the suffix method with the same name. In this sense, the **inner** construct of Beta is less general than the "corresponding" **super** construct found in Smalltalk, but the restriction is justified by the desire for security.

The encoding of a class abstraction supporting a Beta-style class model is given in Figure 8.14 and is a straightforward mapping of $BetaClass$ into PICCOLA(F). Note that we again use the extensibility of the abstraction `Class` in order to define the abstraction `BetaClass` as a simple extension. The encoding of the patterns Person and Graduate immediately follows from the pattern definitions given above and have therefore been omitted here.

8.3.5 Multiple inheritance

As the last abstractions in this section, we briefly discuss the influence of generators and wrappers for defining class abstractions for multiple inheritance. In object-oriented programming languages that support multiple inheritance, multiple parents of a class can have instance variables or method with the same name (also known as the phenomenon of *name collisions*). To deal with these name collisions, various mechanisms have been proposed, but these solutions generally restrain software reusability (refer to [VLM96] for a survey). In this work, we do not intend to discuss all the mechanisms and the related consequences; we only illustrate how the basic ideas of some of these mechanisms can be adapted to the concepts of generators and wrappers.

As a first approach, consider the class abstraction $\mathit{MultipleClass}$ which requires two parent-classes (denoted by P_1 and P_2, respectively):

$$\begin{array}{rcl} \mathit{MultipleClass}(A) & = & \mathit{fix}_{\mathit{Mself}}\ [\ \lambda \mathit{Mself} \rightarrow \\ \{\ G(I) & \rightarrow & \text{let } \mathit{Obj}_{I_1} = A.P_1.G(I),\ \mathit{Obj}_{I_2} = A.P_2.G(I) \text{ in} \\ & & \quad \mathit{Obj}_{I_1} \oplus \mathit{Obj}_{I_2} \oplus A.\Delta(I \oplus \{\mathit{orig}_1 = \mathit{Obj}_{I_1},\ \mathit{orig}_2 = \mathit{Obj}_{I_2}\}), \\ W(I) & \rightarrow & \mathit{fix}_s\ [\ \mathit{Mself}.G(\{\mathit{init} = I,\ \mathit{self} = s\})\ ]\ \}\ \oplus A\ ] \end{array}$$

The Δ requires two parameter $orig_1$ and $orig_2$ in order to separately access the intermediate objects created by both parent-classes. Furthermore, the formalism "hard-wires" that in case of a name collision, the methods of the second parent-class have precedence over methods of the first parent. This scheme is applied by several object-oriented programming languages (e.g. Python [vR96]). However, the formalism does not specify how to proceed in the case of repeated inheritance (i.e. an ancestor class is inherited multiple times).

In the context of subobject-based inheritance, Rossie et. al define the notion of *shared* and *replicated* multiple inheritance [RFW96]. Adapting these notions to our view of inheritance, shared multiple inheritance means that only a single generator-object of a multiply inherited parent-class is created (which is *shared* by the corresponding subclasses), whereas in the replicated case, generator-objects are created multiple times.

A class abstraction for replicated multiple inheritance corresponds to the scheme of the formalism illustrated above and will not be further discussed. In the following, we will explain why a class abstraction for shared multiple inheritance cannot be expressed as a combination of the generators as they have been defined so far and that a splitting of the functionality into *pre-* and *post-*generators is necessary.

Reconsider the class HistoryBoundedPoint as a multiple heir of the class Point, which includes the functionality of both the classes BoundedPoint and HistoryPoint. In order to get the desired behaviour, i) the instance variables x and y and the method move that occurred in the class Point should be shared whereas ii) the two different specializations of the method move should be combined.

The first requirement (sharing of common attributes) could be achieved by calling the generator of the class Point only once and pass this (unique) generator-object to both the generators of BoundedPoint and HistoryPoint. However, this is not possible as a generator is not an abstraction over P. Therefore, we have to split the functionality into a pre- and post-generator: the former is an abstraction over self and a parent generator P_G whereas the latter specifies the binding for the class generator to be used. Hence, the class abstraction $\mathit{MultipleClass}'$ can be defined as follows:

$$\begin{array}{lll} \multicolumn{3}{l}{\mathit{MultipleClass}'(A) \;=\mathit{fix}_{\mathit{Mself}}\;[\;\lambda \mathit{Mself} \rightarrow} \\ \{\; G^{pre}(I) & \rightarrow & \text{let } \mathit{Obj}_I = I.P_G(I) \text{ in } \mathit{Obj}_I \oplus A.\Delta(I \oplus \{\mathit{orig} = \mathit{Obj}_I\}), \\ G^{post}(I) & \rightarrow & \text{let } P_G^M(Y) = A.P_1.G(Y) \oplus A.P_2.G(Y) \text{ in} \\ & & \quad P_G^M(I) \oplus \mathit{Mself}.G^{pre}(I \oplus \{P_G(X) \rightarrow P_G^M(X)\}), \\ W(I) & \rightarrow & \mathit{fix}_s\;[\;\mathit{Mself}.G(\{\mathit{init} = I, \mathit{self} = s\})\;]\;\}\;\oplus A\;] \end{array}$$

Although the abstraction $\mathit{MultipleClass}'$ specifies the same behaviour as $\mathit{MultipleClass}$ (this is ensured by the local abstraction P_G^M defined in the post-generator G^{post}), it is much easier for adaptation to different inheritance models as simply the abstraction P_G^M needs to be redefined in order to reflect an adaptation.

In case of shared multiple inheritance, for example, the abstraction P_G^M of the corresponding class i) parses his ancestor hierarchy in order to find multiple occurrences of the same ancestor class(es), ii) defines an order in which the pre-generators of his ancestor classes have to be called (similar to the computation of the class precedence list in CLOS), and iii) calls the pre-generators with appropriate parameters.

As we already pointed out in section 8.3.2, the required behaviour cannot be achieved by simply calling the Δ abstractions of the parent-classes with appropriate parameters, as Δ only defines the *difference* to the parent-class(es), but does not specify how self-calls have to be dispatched.

The second requirement (combination of the methods move) is not solved by splitting the generator functionality. Its easy to show that the required behaviour of the method move cannot be achieved as simple combination of two parent calls and that at least a part of the functionality of one of the inherited move methods has to be reimplemented in the class HistoryBoundedPoint.

As a summary, we conclude that i) generators and wrappers as they were previously defined do not have enough expressive power to cope with (shared) multiple inheritance and ii) a splitting of the functionality of a generator into a pre- and post-generator only solves some of the problems of multiple inheritance models (e.g. the problem of name collisions is not fully solved). Luckily, as we will discuss in the next section, there is a solution to most of the remaining problems based on the notion of *mixins*.

8.4 Mixins

In order to overcome some of the problems with multiple inheritance, several authors have proposed the notion of *mixins* [BC90, Coo89, VLM96]. The concept of mixins was first introduced in LISP-based languages like Flavors and CLOS as a special use of the already present multiple inheritance mechanism. In this approach, a mixin is regarded as an *abstract subclass*, a class definition that may be applied to different parent-classes to create a related family of modified classes. More specifically, a mixin is a class without specified parent-class and is usually intended to support some aspect of behaviour orthogonal to the behaviour supported by other classes.

From a different point of view, mixins cannot be considered as a special case, but rather as the supporting mechanism for inheritance. Therefore, Bracha and Cook defined the term *mixin-based inheritance* [BC90]. The essence of mixin-based inheritance is to view mixins as stand-alone entities that can be used (or *mixed-in*) for the construction of different classes. In this approach, a class is obtained as a *composition* of mixins and can be considered as an entity which does not refer to its (still unbound) parent-class.

An important difference between mixin-based inheritance and mixins in CLOS is the way parent-classes are merged during inheritance.

- In CLOS, the ordering of this merging is determined by a linearization algorithm, which can be altered by using the CLOS metaobject protocol [KdRB91]. Each ancestor of a given class occurs *only once* in the resulting linearized inheritance graph. This may result in an unexpected and sometimes even non-intuitive behaviour.

- In mixin-based inheritance, a programmer has explicit control over the linearization: the inheritance chain is made *explicit*. This avoids unforeseen insertions of mixins between a class and its parent(s), and the resulting behaviour is hardly unexpected or non-intuitive.

  Explicit control also allows the same mixin to occur more than once in an inheritance chain: the construction of the class DoubleBoundedPoint, for example, requires the mixin Bound to be mixed-in twice (refer to section 8.5).

In this section, we will define the notion of mixins, mixin application, and mixin composition as the main mechanisms of mixin-based inheritance, using the concepts of wrappers and generators. We also discuss different examples of mixin abstractions and show how the classes of our sample hierarchy can be defined using mixin application and composition.

As an extension of the mixin concept, one could think of upgrading linear mixin-based inheritance to *multiple* mixin-based inheritance by providing a mixin with multiple parametrical parent variables. However, such an approach would undo the linear property of mixins and, consequently, reintroduce the same problems multiple inheritance suffers from. Therefore, for the rest of this work, we will only consider mixin abstractions with a single parent parameter.

8.4.1 Basic mixin abstraction

In our model, a mixin is an *abstract subclass* (i.e. a class definition with an unbound parent P). Hence, like classes, mixins are represented as first-class values (i.e. *mixin metaobjects*). The binding of the unbound parent P of a mixin M is established by *applying* mixin M to a class D, written as $M * D$. In fact, applying a mixin M to a class D results in a new class which combines the behaviour of both M and D with precedence of the behaviour of M.

There are several possibilities how mixins can be defined in terms of generators and wrappers, but we have opted for a solution where the generator for a mixin is defined as an abstraction over $self$ and a parent P. Furthermore, in a first approach, a mixin application $M * D$ is encoded as $M.A(D)$. Hence, a mixin abstraction Mixin (supporting Smalltalk semantics) is defined as follows (in order to simplify the formalism, we do not distinguish between a pre- and a post-generator):

$$
\begin{array}{lll}
\mathit{Mixin}(X) & = \mathit{fix}_{\mathit{Mself}}\ [\ \lambda \mathit{Mself} \rightarrow & \\
\quad \{\ G(I) & \rightarrow & \text{let } \mathit{Obj}_I = I.P.G(I) \text{ in } \mathit{Obj}_I \oplus X.\Delta(I \oplus \{\mathit{orig} = \mathit{Obj}_I\}), \\
\quad A(D) & \rightarrow & \mathit{Class}(A \oplus \{P = D\}), \\
\quad W(I) & \rightarrow & \mathit{fix}_s\ [\ \mathit{Mself}.G(\{\mathit{init} = I, \mathit{self} = s, P = I.P\})\ ]\ \}\ \oplus X\ ]
\end{array}
$$

In general, defining a wrapper for a mixin M is useless since a mixin only specifies partial behaviour of objects. However, defining a mixin wrapper as an abstraction over a parent-class P allows us to define instances of *anonymous classes* $M * P$. Anonymous classes are useful if only a single instance is needed. This approach can be compared to anonymous lambda abstractions found in functional programming languages and has been adopted in the wrapper W defined above.

The abstraction `Mixin` illustrated in Figure 8.15 corresponds to a PICCOLA(F) encoding of the abstraction Mixin defined above. Note that the mixin method `Apply` used to encode mixin application (i.e. the method A used above) explicitly calls the abstraction `Class` in order to implement mixin application. Hence, applying mixin metaobject to a class metaobject yields a new class metaobject with a Smalltalk-like behaviour. For mixin abstractions with different method dispatch strategies, the same techniques can be applied as discussed in section 8.3.3 (e.g. moving the fixed-point operator into the generator).

8.4.2 Mixin composition

In the previous section, we have discussed mixin application as the first mechanism of mixin-based inheritance: applying a mixin to a class results in a new class which combines the behaviour of the two abstractions. However, as we have already pointed out in the introduction to mixins, it is possible to take mixins as the primary definitional construct and define inheritance as *mixin composition*. A class is simply viewed as a degenerate mixin that does not refer to its parent parameter and defines all attributes that it refers to in itself. The term *complete* mixin is often used as an acronym for class.

The operator $*$ we have introduced above is in fact nothing else than a general-purpose *mixin composition* operator. Mixin composition takes two mixins M_1 and M_2 and yields a composite mixin $M_1 * M_2$ which combines the behaviour of M_1 and M_2, but gives precedence to the behaviour of M_1. The reader should note that, by definition, mixin application and composition are associative: applying a composed mixin $M_{1,2} = M_1 * M_2$ to a class D, then the resulting class $D' = (M_1 * M_2) * D$ is equivalent to applying M_1 to the application of M_2 to D (i.e. $(M_1 * M_2) * D = M_1 * (M_2 * D)$).

There are several options how mixin composition can be integrated into the abstraction Mixin. We have opted for the solution to use a single concept for both mixin composition and application and defined the concept of a *composer abstraction* C. A composer

```
function Mixin(Args) =
  let
    value MetaSelf = emptyRef()
    function Mself() = MetaSelf.get()

    function MixinIntermediate(Pars) =
      let
        value ParentIntermed = Pars.parent().CreateIntermediate(<Pars>)
      in
        < {- polymorphic form extension -}
          ParentIntermed,
          Args.Delta(<Pars, orig = ParentIntermed>)
        >
      end

    function Create(Init) =
      let
        value Self = emptyRef()
        function self() = Self.get()
        value NewInstance = Mself().CreateIntermediate (
                              <init=Init, self=self, parent=Init.parent()>)
      in
        Self.set(<NewInstance>);
        NewInstance
      end

    function Apply(Parent) =                      {- mixin application -}
        Class(<Args, parent = Parent>)            {- explicit Class call -}

    value MixinInstance =
      < {- mixin meta abstractions -}
        Apply = Apply,
        CreateIntermediate = MixinIntermediate,

        {- other meta abstractions -}
        Create = Create,

        {- user-defined parameters and methods -}
        Args
      >
  in
    MetaSelf.set(<MixinInstance>);
    MixinInstance
  end
```

Figure 8.15: Mixin abstraction with dynamic method dispatch.

abstraction is a meta-level operation for both class and mixin metaobjects and takes as argument the left-hand side operand of a mixin application or composition. Hence, $M * D$ is modelled as $D.C(M)$ whereas $M_1 * M_2$ is encoded as $M_2.C(M_1)$. This approach allows us to omit a polymorphic class method A and the corresponding explicit call of the abstraction $Class$.

Both mixin application and composition require a distinguished composer C. In case of the abstraction $Class$, the composer C is defined as follows:

$$\begin{array}{lcl} C(M) & = & \text{let } G^{comp}(X) = M.G(X \oplus \{P = \mathit{Mself}\}) \text{ in} \\ & & \quad Class\ (\{G(I) \rightarrow G^{comp}(I)\}) \end{array}$$

Note that the composer abstraction C creates a new class metaobject by solely passing a generator G to the abstraction $Class$. This is possible as the corresponding protocol i) is based on keyword-based parameter passing and ii) enables overriding of the default generator behaviour.

The code of the method `Compose` for `Class` corresponds to the formalism defined above: it i) creates an new generator `ComposedIntermediate` based on its own generator and the generator of the mixin it is applied to and ii) instantiates an new class metaobject using the new generator (there remaining code for `Class` is unchanged and corresponds to the code shown in Figure 8.10):

```
function Compose(Other) =
  let
    function ComposedIntermediate(Pars) =
        Other.CreateIntermediate (<Pars, parent = Mself()>)
  in
    Class (< CreateIntermediate = ComposedIntermediate >)
  end
```

In case of mixin composition $M_1 * M_2$, the situation is different. In the composite mixin, the parent of M_1 has to refer to M_2 whereas the parent of M_2 will be bound to a class, say D, by a final mixin application (e.g. $(M_1 * M_2) * D$). Therefore, the abstraction $Mixin$ is adapted as follows:

$$\begin{array}{l} Mixin(A) \ = \mathit{fix}_{\mathit{Mself}}\ [\ \lambda \mathit{Mself} \rightarrow \\ \quad \begin{array}{rcl} \{\ G(I) & \rightarrow & \text{let } Obj_I = I.P.G(I) \text{ in } Obj_I \oplus A.\Delta(I \oplus \{orig = Obj_I\}), \\ C(M) & \rightarrow & \text{let } G^{comp}(X) = M.G(X \oplus \{P = X.P.C(\mathit{Mself})\}) \text{ in} \\ & & \quad Mixin\ (\{G(I) \rightarrow G^{comp}(I)\}), \\ W(I) & \rightarrow & \mathit{fix}_s\ [\ \mathit{Mself}.G(\{init = I, self = s, P = I.P\})\ ]\ \}\ \oplus A] \end{array} \end{array}$$

The PICCOLA(F) code for `Mixin` is a straightforward encoding of the abstraction $Mixin$ defined above and has therefore been omitted here.

The reader should note that mixin composition (and application) based on composition of generators behaves correctly for any method dispatch strategy (e.g. if a mixin M_1 uses static method dispatch, also its composition with a mixin M_2 and its application to a class D uses static method dispatch). This is not the case if mixin composition (and application) is based on Δ abstractions alone.

HistoryPoint	=	History $*$ Point
BoundedPoint	=	Bound $*$ Point
HistoryBoundedPoint	=	(History $*$ Bound) $*$ Point
LinearBoundedPoint	=	(Linear $*$ Bound) $*$ Point

Table 8.1: Point class hierarchy using mixin application and composition.

8.4.3 Examples revisited

The abstraction `Mixin` allows us to define all subclasses of the class Point (except the class DoubleBoundedPoint) using mixin application and composition. In order to do so, we implement the mixins History, Bound, and Linear. The mixin History is defined as follows (the other two mixins are similar):

```
value History = Mixin (
   <
      function Delta(Pars) =
         let
            function self() = Pars.self()
            value orig = Pars.orig()
         in
            <
               function move(Args) =
                   PrVal(<self().X()>); PrVal (<self().Y()>);
                   orig.move(<Args>)
            >
         end
   >
)
```

Note that the method `move` does not need to access the formal argument `Args` and simply passes it on to the method `move` of the parent-class.

Given the class Point as defined in section 8.3.2, the class HistoryPoint is a simple application of the mixin History to Point. For the class HistoryBoundedPoint, it is necessary to decide which of the to mixins History and Bound should have precedence: due to the fact that mixin composition is not commutative, the order of the mixins matters and the resulting classes do not have the same behaviour. For the rest of this work, we assume that the mixin History has precedence over the mixin Bound. A summary of the resulting mixin applications for the point class hierarchy is shown in Table 8.1.

Note that the class DoubleBoundedPoint cannot be defined using the mixin abstractions we have illustrated so far: it requires an *encapsulation operator* which will be discussed in section 8.5.

8.4.4 Singleton mixin

Sometimes it is desirable (or even necessary) to have only one instance of a particular class. As an example, consider a single window manager for a windowing environment. In such a situation, this class has to fulfill the specification of a *singleton* behaviour, which has been described as a pattern in [GHJV95]. The *Singleton* pattern makes a class responsible for keeping track of its sole instance, providing the only access point to that instance, and preventing the creation of multiple instances.

Several variations of the Singleton pattern have been described (refer to [GHJV95] or [ABW98] for details), but all these descriptions assume that the singleton class has to be designed from scratch. If this is not the case (i.e. a class defining the desired functionality already exists, but it does not specify a singleton behaviour), it is necessary to define a glue abstraction which adapts the class accordingly.

Another problem of existing solutions is that a singleton class requires special usage, as the access point for the singleton instance is in general not a simple constructor call (e.g. the solution described in [GHJV95] uses a static class method).

In Figure 8.16, a mixin `Singleton` is shown which i) defines singleton behaviour and ii) overcomes the two problems discussed above. More precisely, the mixin `Singleton` defines a *generic* glue abstraction which cannot only be applied to classes, but also to mixins. In both cases, the resulting abstraction (either a class metaobject or a mixin metaobject) has the required behaviour. In order to treat singleton classes like any other class, instances can be created by calling the method `Create`. However, the implementation of the corresponding generator ensures that a call to `Create` always returns a reference to the same object. Therefore, the access point to the singleton instance is simply the constructor of the corresponding class.

The generator G of the mixin $Singleton$ can be formalized as follows:

$$G(I) \quad = \quad \Diamond\,[\; I.P.G(I)\;]$$

Applying the operator $\Diamond$ to a function f implies that f behaves like a function that always returns the result of the first function call (similar to a **once** function in Eiffel [Mey92]). The reader should note that the concept of a **once** function, that is not state-independent, does not exist in a pure functional formalism. There exist a variety of approaches to introduce state in a pure functional formalism, but the concept of *monads* is probably the best-known [Mog89, Wad92]. In the FORM calculus, however, an abstraction to generate a **once** function can be easily defined using *reference cells* (refer to Figure 8.16 for details).

The composer abstraction C of the mixin $Singleton$ is formalized as follows:

$$\begin{array}{rcl} C(M) & = & \text{let } G^{comp}(X) = \Diamond\,[M.G(X \oplus \{P = X.P.C(Mself)\})\,] \text{ in} \\ & & \quad Mixin\;(\{G(I) \rightarrow G^{comp}(I), C(M) \rightarrow C(M)\}) \end{array}$$

Note that the mixin `Singleton` defined in Figure 8.16 is not a mixin abstraction, but a single value (i.e. there is only one mixin "instance" available): it is the only instance of a locally defined mixin abstraction `TSingleton`. `TSingleton` can be considered as an adaptation of the abstraction `Mixin`: it overrides the method `Compose`. Internally, the mixin `Singleton` defines a abstraction `SingletonIntermediate` which defines a singleton-generator factory and encodes the $\diamond$ operator. In order to simplify the encoding, the method `Create` of `TSingleton` has been omitted in Figure 8.16.

Another problem with existing solutions appear in the context of inheritance. Our approach ensures that if a class C' is a subclass of a singleton class C, all instances of C' share a single generator-object of C. The behaviour defined in Δ of C' is not shared due to the inheritance model we use.

However, using a the `Singleton` mixin as the right-hand side argument of a mixin composition, the resulting mixin preserves the singleton behaviour. This is due to the definition of the composer C which i) applies a $\diamond$ operator to the composite generator G^{comp} and ii) overrides the default composer behaviour of the abstraction $Mixin$.

As a brief summary, we conclude that singleton behaviour can be expressed as an application of a $\diamond$ operator to a composition of generators. In contrast to other meta-level approaches for singleton behaviour, where a singleton class has to be an instance of specific singleton *metaobject class*, the approach using mixins does not have such a requirement and, therefore, enhances the reuse of existing classes. This approach also conforms to the intention of mixins to support some aspect of behaviour orthogonal to the behaviour supported by other classes or mixins.

8.5 Applying form restriction

The encodings of the class and mixin abstractions defined in the previous sections could have also been modelled in the $\pi\mathcal{L}$-calculus, since we did not use form restriction nor matching. In this section, however, our modellings go beyond the $\pi\mathcal{L}$-calculus: we present the encoding of an encapsulation operator which heavily depends on polymorphic form restriction.

In the literature, the term *encapsulation* has been employed with different meanings. For the rest of this work, encapsulation is used for *attribute hiding*. Encapsulating attributes is generally accepted as an important mechanism of object-oriented software engineering. Due to a conflict of interests, the interaction between encapsulation and incremental modification (inheritance, mixin composition and application) is very delicate. On one hand, a subclass should have the possibility to override the implementation attributes of its parent-class whereas on the other hand, a subclass should not have access to the implementation details of its parent.

In software engineering it is desirable to have the opportunity to write the code of a class and its parent-class(es) independently (after having specified the necessary interfaces and interaction protocols). However, the use of protected and private attributes (as it is available in the programming language C++) exhibits a conflict between data encapsulation and reusability interests [VLM96]. Therefore, a more general mechanism is needed that allows the possibility to restrict the visibility of attributes towards the interiors of a

```
value Singleton =
  let
    value MetaSelf = emptyRef()
    function Mself() = MetaSelf.get()

    function SingletonIntermediate() =          {- helper abstraction -}
      let
        value first = ref(<true>)
        value single = emptyRef()
      in
        < function CreateIntermediate(Pars) =
            if first.get() then
              single.set(<Pars.parent().CreateIntermediate(<Pars>)>);
              first.set(<false>); single.get()
            else
              single.get()
            end >
      end

    function Apply(Other) = Other.Compose(<Mself()>)

    function SingletonCompose(Pars) =          {- singleton composition -}
      let
        value TmpIntermediate = SingletonIntermediate()
        function ComposedIntermediate(Other) =
          let
            value MPI=Pars.parent().Compose(<Mself(),TmpIntermediate>)
          in
            Other.CreateIntermediate(<Pars, parent = MPI>)
          end
      in
        Mixin (<CreateIntermediate = ComposedIntermediate,
              Compose = SingletonCompose>)
      end

    value SingletonInstance =
      <
        SingletonIntermediate(), Compose = SingletonCompose,
        Apply = Apply
      >
  in
    MetaSelf.set(SingletonInstance);
    SingletonInstance
  end
```

Figure 8.16: Source code of singleton mixin.

class. In the following, we will discuss the notion of an *encapsulation operator* (originally defined in [VLM96]) and show how it can be integrated into our FORM calculus-based object model.

The encapsulation abstraction defined in our model can be seen as a variant of the *hide* operator presented by Bracha and Lindstrom [BL92]. It ensures that encapsulated methods become invisible to any client of a class or mixin encapsulation is applied to. Intuitively, encapsulating a method foo of a class D consists of i) replacing dynamically dispatched self-calls to foo by statically dispatched calls and ii) removing foo from the intermediate object created by the corresponding generator.

The abstraction $Encapsulate$ defined below is similar to the abstraction $Mixin$ presented previously: it is only an abstraction over Δ. In fact, a metaobject created by $Encapsulate$ (i.e. an *encapsulation metaobject*) has to be considered as a mixin metaobject and, therefore, can be applied to classes or composed with other mixins. However, in contrast to the class and mixin abstractions, the purpose of Δ is different as it does not specify an incremental behaviour, but the set of methods to be encapsulated.

$$
\begin{array}{rcl}
\multicolumn{3}{l}{Encapsulate(A) \;=\; \mathit{fix}_{Mself}\;[\;\lambda Mself \rightarrow} \\
\{\; G(I) & \rightarrow & \mathit{fix}_s\;[\text{let } self' = I.self \oplus s,\; Obj_I = A.P.G(I \oplus \{self = self'\}) \text{ in} \\
& & \qquad\qquad Obj_I \setminus A.\Delta(I \oplus \{orig = Obj_I,\; self = self'\})] \\
C(M) & \rightarrow & \text{let } G^{comp}(X) = M.G(X \oplus \{P = X.P.C(Mself)\}) \text{ in} \\
& & \qquad Mixin\;(\{G(I) \rightarrow G^{comp}(I)\}), \\
W(I) & \rightarrow & \mathit{fix}_s\;[\;Mself.G(\{init = I,\; self = s,\; P = I.P\})\;]\;\}\;\oplus A]
\end{array}
$$

In order to avoid interference with the mixin composition operator $*$, only attributes not referring to `orig` are allowed to be encapsulated (e.g. methods not performing a orig-call). This ensures that encapsulated attributes cannot interfere with attributes of a parent-class. Similarly, since all self-calls are statically dispatched, encapsulated attributes cannot interfere with "child" attributes. Therefore, encapsulation and inheritance/mixin composition can be considered as *orthogonal* mechanisms.

Originally, the encapsulation operator is defined as an additional primitive to a formalism for mixin-based inheritance [VLM96]. However, as the code shown in Figure 8.17 illustrates, it can be easily encoded in the FORM calculus using both, polymorphic form extension and restriction.

Using the encapsulation operator illustrated above, it is possible to define the class DoubleBoundedPoint as combination of encapsulation and mixin application. The expression Encaps (Linear $*$ Bound, {bound}) encapsulates the attribute bound from the composition of the mixins Linear and Bound.

$$\text{DoubleBoundedPoint} \;=\; \text{Bound} * \text{Encaps}\;(\text{Linear} * \text{Bound},\; \{\text{bound}\}) * \text{Point}$$

This illustrates that the mixin Bound is applied twice, once specialized with the Linear behaviour.

```
function Encapsulate(Args) =
  let
    function EncapsIntermediate(Pars) =
      let
        value newSelf = emptyRef()           {- local self reference -}
        function self() = <Pars.self(), newSelf.get()>

        value ParentIntermed =
             Args.parent().CreateIntermediate(<Pars,self=self>)
        value EncapsPI = ParentIntermed \ Args.Delta()
        value EncapsSelf = ParentIntermed \ EncapsPI
      in
        newSelf.set(<EncapsSelf>);
        PrForm(<EncapsPI>);
        EncapsPI
      end
  in
    Mixin(<CreateIntermediate = EncapsIntermediate>)
  end
```

Figure 8.17: Source code of encapsulation abstraction.

8.6 A meta-level framework

In the previous sections, we have defined meta-level abstractions for modelling various concepts found in object-oriented programming. In this section, we define generalization of all meta-level abstractions and specify a meta-level framework for concurrent, object-oriented programming.

Analyzing the class abstractions defined in the previous sections, we argue that generators and wrappers define the *semantic model* (i.e. the underlying inheritance model, the method-dispatch strategy etc.) for a related family of classes whereas Δ and the collection of parent-class generators specify the behaviour of a concrete class. Furthermore, we can see that generators and wrappers also specify a *meta-level protocol*: the generator of a class metaobject, for example, creates intermediate object(s) of its parent-class(es) and the intermediate object of Δ, and composes these objects according to the given semantic model. The way these intermediate objects are composed is different for each semantic model (e.g. in a Smalltalk-like model, methods of a class have precedence over the methods defined in a parent-class whereas in a Beta-style model, the parent-class methods have precedence), but the "ingredients" of the composition are the same for all semantic models. Hence, it is possible to split the functionality of generators into a static protocol-part and a variable model-part, denoted as *concrete* and *model* generators, respectively. Therefore, it is sufficient to define only one concrete generator and parameterize it with appropriate model generators. A similar separation can also be achieved for wrappers.

Driven by our motivation to generalize the definition of concurrent, object-oriented abstractions and the observations described above, we have defined the meta-level frame-

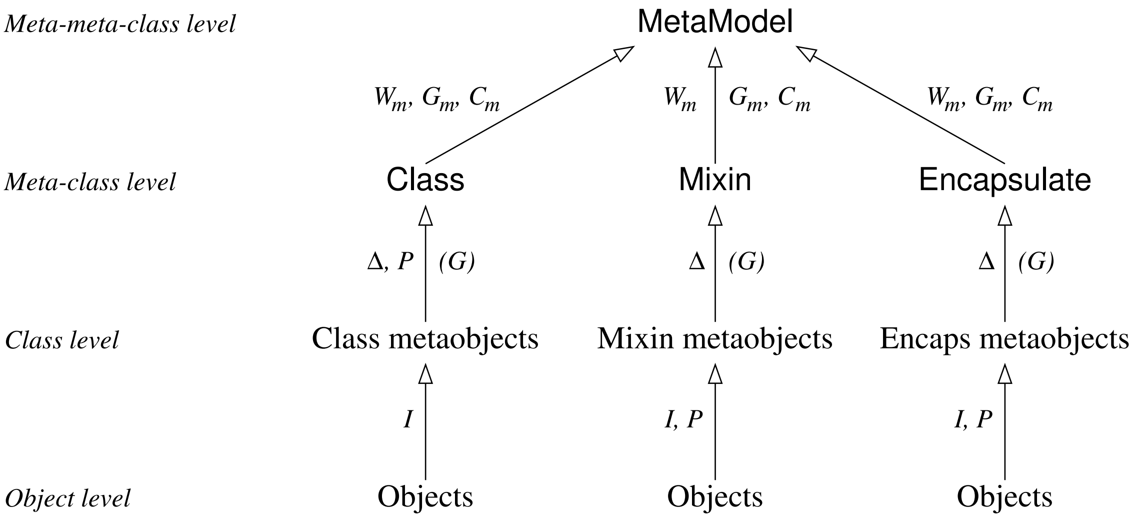


Figure 8.18: Conceptual view of the meta-level framework.

work illustrated in Figure 8.18, which defines a hierarchy of meta-level abstractions. Arrows between entities at different levels denote inter-level dependencies whereas the annotation on the arrows represent the formal arguments of the corresponding dependencies.

On top of the hierarchy is the MetaModel abstraction which defines the generic framework for class abstractions and metaobjects. It provides the common behaviour of all class and mixin abstractions, the corresponding meta-protocol, and in particular the concrete generators, wrappers, and composers. In fact, the MetaModel abstraction is a *meta-class* abstraction (i.e. it is an abstraction to create class abstractions).

The abstractions Class, Mixin, and Encapsulate illustrated in the previous sections define the meta-behaviour for a specific semantic model and are instantiated by passing appropriate model generators, model wrappers, and model composers to the MetaModel abstraction. Class, mixin, and encapsulated metaobjets are created by passing a Δ abstraction and a (possibly empty) collection of parent-class metaobjects, written P, to a meta-class abstraction.

The reader should note that MetaModel and the class and mixin abstractions are represented as *functions* whereas the class- and object-level entities are objects (i.e. record-like structures). Hence, unlike other meta-level approaches (e.g. Smalltalk or CLOS), class metaobjects are not instances of class metaobject classes.

8.6.1 The MetaModel abstraction

The MetaModel abstraction which specifies the common meta-protocol for meta-class level abstractions is defined as follows:

$$MetaModel(X) \quad = \quad \lambda A \rightarrow \mathit{fix}_{Mself}\; [\; \lambda Mself \rightarrow Static\text{-}Protocol\text{-}Part \oplus A \;]$$

$MetaModel$ is a function that expects a keyword-based argument X with bindings for G_m, W_m, and C_m, denoting the model generator, model wrapper, and model composer to instantiate a meta-class level abstraction (e.g. Class) for a concrete semantic model. The result of $MetaModel$ is itself a function, which expects a keyword-based argument A with appropriate bindings for the concrete semantic model (e.g. Δ and P for Class). Evaluating this function yields a class-level object with self-reference $Mself$. This class-level object is defined as a polymorphic extension of the form $Static\text{-}Protocol\text{-}Part$, which defines the static protocol part of the meta-level framework, with the keyword-based argument A passed to the meta-class abstraction. This application of polymorphic form extension enables overriding of the default generator, wrapper, and composer behaviour and is needed for modelling mixin application and composition. The form $Static\text{-}Protocol\text{-}Part$ has the following structure:

$$
\begin{array}{l}
Static\text{-}Protocol\text{-}Part \;=\\
\qquad \begin{array}{lcl}
\{\; G(I) & \rightarrow & X.G_m(I \oplus A)\\
\;\; W(I) & \rightarrow & \mathit{fix}_s\,[\; X.W_m(\{init = I,\; self = s,\; G(Y) \rightarrow Mself.G(Y)\})\;]\\
\;\; C(M) & \rightarrow & MetaModel(X)(\{G(I) \rightarrow X.C_m(\{mixin = M,\; init = I,\\
 & & \qquad\qquad\qquad\qquad Mself = Mself\}\;)\;\})\;\}
\end{array}
\end{array}
$$

The default wrapper W passes i) the init-parameter I, ii) self, and iii) the generator denoted by $Mself.G$ to the model wrapper W_m, and establishes the correct binding of self for the intermediate object created by W_m. In order to avoid interference of the three parameters and to keep things separate, we add structure to the parameter passed to W_m by using nested forms. Note that we use dynamic method binding in W for G in order to reflect the fact that the default generator may have been overridden by a composer C (see below).

The default generator G polymorphically extends the argument passed to itself with the bindings for the concrete semantic model and calls the model generator G_m with the resulting form. Finally, the composer C creates a new metaobject based on the model abstractions passed to $MetaModel$ and a composite generator defined by a model composer C_m. Again, nested forms are used to add additional structure to the argument passed to C_m.

The PICCOLA(F) encoding of $MetaModel$ is illustrated in Figure 8.19. Due to the fact that i) PICCOLA(F) does not support anonymous abstractions and ii) functions are not first-class values (i.e. it is not possible to directly return a function as the the result of another function), the abstraction `MetaModel` defines a local abstraction `MetaClass` and returns this function wrapped up in a form (i.e. it can be accessed using the label `fun`).

8.6.2 Model abstractions for classes

In this section, we will discuss model wrappers, generators, and composers for defining class abstractions for class metaobjects. However, we will only consider selected model abstractions which define single inheritance class models.

```
function MetaModel(MetaPars) =
  let
    function MetaClass(Args) =
      let
        value MetaSelf = emptyRef()
        function Mself() = MetaSelf.get()

        function DefIntermediate(Pars) =
            MetaPars.CreateIntermediate(<Pars, Args>)

        function DefCreate(Init) =
          let
            value Self = emptyRef()
            value Newinstance = MetaPars.Create(<init = Init,
                    CreateIntermediate = Mself().CreateIntermediate,
                    self = Self.get>)
          in
            Self.set (<NewInstance>);
            NewInstance
          end

        function DefCompose(Other) =
          let
            function ComposedIntermediate(Pars) =
                MetaPars.Compose(<other=Other,pars=Pars,Mself=Mself>)
          in
            MetaModel(MetaPars).fun(<
                          CreateIntermediate=ComposedIntermediate>)
          end

        value MetaObjInstance =
          < {- meta abstraction:  generator -}
            CreateIntermediate = DefIntermediate,

            {- meta abstractions:  wrapper and composer -}
            Create = DefCreate,
            Compose = DefCompose,

            {- userdefined parameters and methods -}
            Args
          >
      in
        MetaSelf.set(MetaObjInstance);
        MetaObjInstance
      end
  in
    < fun = MetaClass >
  end
```

Figure 8.19: Source code of the *MetaModel* abstraction.

Common to all (single-inheritance) class abstractions is that they require a Δ abstraction and a parent-class metaobject P as a parameter and that they share the same class model wrapper W_m^C and class model composer C_m^C. The two class model abstractions are defined as follows:

$$\begin{aligned} W_m^C(I) &= I.G\ (\{init = I.init, self = I.self\}) \\ C_m^C(M) &= M.mixin.G\ (M.init \oplus \{P = M.Mself\}) \end{aligned}$$

The main difference between the class abstractions for various class models lies in the way the model generators are defined:

$$\begin{aligned} G_m^{Small}(I) &= \text{let } Obj_I = I.P.G(I) \text{ in } Obj_I \oplus I.\Delta(I \oplus \{orig = Obj_I\}) \\ G_m^{Beta}(I) &= \text{let } Obj_I = I.\Delta(I.P.\Delta_{Inner}() \oplus I.\Delta_{Inner}() \oplus I) \text{ in} \\ &\qquad Obj_I \oplus I.P.G(I \oplus Obj_I) \\ G_m^{Stat}(I) &= fix_{self}\ [\text{ let } Obj_I = I.P.G(I \oplus \{self = self\}) \text{ in} \\ &\qquad Obj_I \oplus I.\Delta(I \oplus \{self = self, orig = Obj_I\})\ ] \end{aligned}$$

The model generator G_m^{Small} defines a Smalltalk-like class model (i.e. dynamic binding of self-calls, direct invocation of inherited methods etc.), G_m^{Beta} defines a Beta-style class model (i.e. a prefix-style of inheritance), and G_m^{Stat} defines a model generator for static method dispatch. Model generators for other schemes like the abstraction $C{+}{+}Class$ can be defined similarly.

Using G_m^{Small}, G_m^{Beta}, G_m^{Stat}, W_m^C, and C_m^C, the class abstractions $Class$, $BetaClass$, and $StaticClass$ can now be defined as follows:

$$\begin{aligned} Class(A) &= MetaModel\ (\ \{G_m(I) \rightarrow G_m^{Small}(I), W_m(I) \rightarrow W_m^C(I), \\ &\qquad C_m(M) \rightarrow C_m^C(M)\}\ )(A) \\ BetaClass(A) &= MetaModel\ (\ \{G_m(I) \rightarrow G_m^{Beta}(I), W_m(I) \rightarrow W_m^C(I), \\ &\qquad C_m(M) \rightarrow C_m^C(M)\}\ )(A) \\ StaticClass(A) &= MetaModel\ (\ \{G_m(I) \rightarrow G_m^{Stat}(I), W_m(I) \rightarrow W_m^C(I), \\ &\qquad C_m(M) \rightarrow C_m^C(M)\}\ )(A) \end{aligned}$$

The PICCOLA(F) encoding of the modified class abstractions illustrated above and the corresponding model generators, model wrappers, and model composers is straightforward and has therefore been omitted here.

8.6.3 Model abstractions for mixins and encapsulation

In this section, we will discuss the model abstractions for both mixins and the encapsulation abstraction. The model wrapper for both kinds of abstractions is defined as follows:

$$W_m^M(I) \;=\; I.G\;(\;I \oplus \{P\!=\!I.init.P\}\;)$$

Both mixin application and composition require a distinguished model composer C_m. In case of mixin application $M * D$, the model composer C_m^C is defined as shown in section 8.6.2. C_m^C returns an intermediate object by passing D as parent (denoted by P) to the generator of M.

As mentioned in section 8.4.2, the situation is different in case of mixin composition and the model composer C_m^M for mixin composition has to defined as follows in order to reflect the correct interpretation of the parent P in all mixins involved:

$$\begin{aligned} C_m^M(M) \;&=\; \text{let } \mathit{MetaObj}\!=\!M.init.P.C(M.\mathit{Mself}) \text{ in} \\ &\qquad M.mixin.G\;(M.init \oplus \{P\!=\!\mathit{MetaObj}\}) \end{aligned}$$

Within C_m^M, the subexpression $M.init.P.C(M.\mathit{Mself})$ creates a new class metaobject $\mathit{MetaObj}$ which defines the application of the right-most mixin to the class metaobject the composite mixin will be applied onto (e.g. applying $(M_1 * M_2)$ to a class metaobject D, then $\mathit{MetaObj}$ denotes $M_2 * D$). This class metaobject is used to bind the unbound parent in the left-most mixin.

Using the model generator G_m^{Small} defined in section 8.6.2, the mixin model composer C_m^M, and the mixin model wrapper W_m^M, the abstraction Mixin (supporting Smalltalk semantics) is defined as:

$$\begin{aligned} \mathit{Mixin}(A) \;&=\; \mathit{MetaModel}\;(\;\{G_m(I) \to G_m^{\mathit{Small}}(I), W_m(I) \to W_m^M(I), \\ &\qquad C_m(M) \to C_m^M(M)\}\;)(A). \end{aligned}$$

The model generator G_m^{Encaps} of the encapsulation abstraction $\mathit{Encapsulate}$ is defined as follows:

$$\begin{aligned} G_m^{\mathit{Encaps}}(I) \;&=\; \mathit{fix}_{\mathit{self}'}\;[\;\text{let } \mathit{Obj}_I\!=\!I.P.G(I\oplus\{\mathit{self}\!=\!I.\mathit{self}\oplus \mathit{self}'\}) \text{ in} \\ &\qquad \mathit{Obj}_I \setminus I.\Delta(I\oplus\{\mathit{orig}\!=\!\mathit{Obj}_I, \mathit{self}\!=\!I.\mathit{self}\oplus \mathit{self}'\})\;] \end{aligned}$$

Using the encapsulation model generator G_m^{Encaps}, the mixin model composer C_m^M, and the mixin model wrapper W_m^M, the abstraction $\mathit{Encapsulate}$ can now be defined as follows:

$$\begin{aligned} \mathit{Encapsulate}(A) \;&=\; \mathit{MetaModel}\;(\;\{G_m(I) \to G_m^{\mathit{Encaps}}(I), W_m(I) \to W_m^M(I), \\ &\qquad C_m(M) \to C_m^M(M)\}\;)(A) \end{aligned}$$

Note that the abstraction $\mathit{Encapsulate}$ uses the same model composer and model wrapper as have been used for the abstraction Mixin.

The PICCOLA(F) encoding of the abstractions Mixin, $\mathit{Encapsulate}$, and the corresponding model generators, model wrappers, and model composers is straightforward and has therefore been omitted here.

8.7 Form introspection

For the encodings of the abstractions we have previously defined in this chapter, we have used some higher-level abstractions (e.g. functions, nested forms) which can be defined in terms of the primitives offered by the FORM calculus. However, there are situations where the expressive power of the FORM calculus is not enough and expressions beyond the calculus are needed. In this section, we will briefly discuss situations in the context of *form introspection*. In particular, we illustrate a mixin abstraction which cannot be expressed using the simple form of introspection offered by matching: information about *all* labels of a given form is required.

Defining a new calculus based on the FORM calculus where form introspection can be encoded in terms of the primitives of the calculus is beyond the scope of this work. However, we will outline some ideas going into this direction in chapter chapter 11.

In the following, we will use the abstractions `labels`, `project`, and `extend` which cannot be defined in the FORM calculus itself, but are implemented as extensions of the PICCOLA(F) run-time system.[9] Note that both `project` and `extend` are generally used in contexts where the labels of a form are given as the result of a `labels` call.

- The function `labels` returns a list with a string representation of all the labels bound by a given form.
- The function `project` can be used to project a label of a form which is given as a string (i.e. `project(<form=F,label="l">)` is equal to `<val=F.l>`).
- The function `extend` can be used to extend a form with an additional binding (i.e. `extend(<form=F,label="l",val=x>)` is equal to `<F,l=x>`). Again, the label `l` is given as a string.

The reader should note that lists can be easily encoded as nested forms, but the concrete encoding is not needed for the following discussion.

As an example where information about all labels of a given form is required, consider the abstraction `Tracing` illustrated in Figure 8.20. The purpose of this abstraction is to trace the sequence of method invocations of an object. The tracing can be turned on or off by using the methods `on` and `off`, respectively. In order to simplify the example, a traced object only prints the methods being invoked.

Similar to the singleton behaviour discussed in section 8.4.4, the corresponding functionality is implemented as a generic mixin. For each exported method defined in the parent-class, the mixin creates a method wrapper with the appropriate functionality (similar to the method wrappers discussed in section 8.2). In order to do so, the method `CreateIntermediate` instantiates an intermediate object `Intermed`, gets a list of all labels bound by `Intermed`, and recursively applies the function `extract` to the list of labels. Applying the mixin `Tracing` to a class metaobject does not affect the interface of its instances; only the behaviour is affected.

[9] Implementing these abstractions requires knowledge about the internal representation of forms.

The function `fold` used in `CreateIntermediate` applies a function given by the label `fun` in a sequential order to all elements of a list. However, in contrast to the function `map` (which applies a function to all elements of a list and returns the resulting list), the result is accumulated over all elements. Therefore, the result of `fold` is generally not a list, but a single value.

Note that the code defined in the method `CreateIntermediate` is generic as it does not specify any specific behaviour needed for tracing; the tracing behaviour is defined in the abstraction `CreateWrapper` which is used by the function `extract`. The corresponding behaviour for tracing is defined as follows:

```
function CreateWrapper(Args) =
  let
    function wrapper(Pars) =
      if Args.form().tracing() then
        PrVal (<concat (<lval = "Calling traced function:",
                         rval = Args.label>)>)
      end;
      project(<form = Args.form, label = Args.label>).val (Pars)
  in
    extend(<form = <>, label = Args.label, val = wrapper>)
  end
```

`CreateWrapper` requires an intermediate object of the corresponding class (given by the label `form`) and the name of the method to be called (given by `label`). It creates a new function `wrapper` which i) prints the name of the method that is invoked (if tracing is turned on) and ii) calls the original method. As the result, it returns a form which binds a label with the same name as `label` passed to the function `wrapper`. Note that the function `wrapper` uses a library abstraction `concat` to concatenate two strings.

Analyzing the structure of the mixin `Tracing`, we can identify a *generic* part (i.e. the method `CreateIntermediate` which adds a method wrapper to each exported method) and a *specific* part (i.e. the abstractions `CreateWrapper` and `Delta` which define the concrete behaviour of the method wrappers). Therefore, it is possible to abstract from the abstraction `Tracing` and define a generic abstraction for *generating* mixins which add method wrappers to objects. We will use the term *wrapin* for these kind of mixins in the following.

Due to the encoding of the abstraction `Class`, it is not possible any more to add synchronization abstractions (such as GSPs) in the same way it is described in section 8.2. However, by using appropriately parameterized method wrappers, we can use a wrapin abstraction to achieve the same behaviour.

Furthermore, the example of the abstraction `Tracing` shows that obtaining information about the labels of a given form is an important feature for modelling particular kinds of compositional abstractions, but it is generally not enough. It is also necessary to use this information to either access specific labels of the corresponding form or to define/extend a form with bindings for some of the retrieved labels. Therefore, all three extensions are essential, and the abstraction `labels` can generally not be used in isolation.

```
value Tracing =
  let
    function TraceDelta(Pars) =
      let
        value trace = ref(<true>)
      in
        <
          function on(Args) = trace.set (<true>),
          function off(Args) = trace.set (<false>),
          function tracing(Args) = trace.get()
        >
      end

    function TraceIntermediate(Pars) =
      let
        value ParentIntermed = Pars.parent().CreateIntermediate(Pars)
        value Intermed = < ParentIntermed, TraceDelta(<>) >
        value IntermedLabels = labels(<Intermed>)

        function extract(Args) =
          <
            Args.res(),
            CreateWrapper(<label = Args.item().val, form = Intermed>)
          >
      in
        fold(<list = IntermedLabels, fun = extract, res = <>>)
      end
  in
    Mixin(<Delta=TraceDelta, CreateIntermediate=TraceIntermediate>)
  end
```

Figure 8.20: Source code of the `Tracing` abstraction.

8.8 Comparison with other meta-level approaches

In the previous sections we have shown that object-oriented features such as inheritance, dynamic binding, self-reference etc. can be conveniently modelled in the FORM calculus with the aid of agents representing meta-level abstractions. There are other object-oriented programming languages taking a similar approach, and where classes are first-class entities at run-time.[10] In this section, we will compare our FORM calculus-based meta-level framework for objects and classes with the corresponding models of CLOS [KdRB91], Smalltalk [GR89], and Python [vR96]. However, it is not our goal to give a detailed overview of these languages (see the corresponding references for details), but to compare the properties of the FORM calculus-based object model with similar features found in these languages.

[10] In statically compiled languages such as C++ or Eiffel, classes are only available at compile-time.

8.8.1 Structure of classes

Common to the FORM calculus-based object model and the three languages we consider is the fact that i) a *class definition* is used to specify the structure (i.e. the data elements) and/or the behaviour (operations on data elements) of objects and ii) classes are represented as first-class entities (i.e. *classes as objects*) of a running application. Furthermore, all class models define a root of the class hierarchy which is a common ancestor to all (base-level) classes. This root class is often needed to define some default behaviour for all instances and to break the recursion of method lookup.

In Smalltalk, a class definition specifies both the data elements and the corresponding operations (*variables* and *methods* in Smalltalk terminology). A class can be defined by sending the message `subclass` to an already existing class, which automatically makes this class the *superclass* of the new class to be defined.

Each class `C` is the only instance of a corresponding *metaclass* (named `C class`) which defines the behaviour of the class. The metaclass of a class is automatically generated when the class is defined. If a class `C` is a subclass of a class `D`, then the same relationship also holds for the metaclasses (i.e. `C class` is a subclass of `D class`). The class `Object` defines the root of the class hierarchy (i.e. it is the only class without a superclass) and provides default behaviour common to all objects.

Due to fact that a metaclass is again a class, the metaclass `Object class` is an instance of the metaclass `Metaclass`. Furthermore, `Metaclass` is an instance of `Metaclass class`. In order to break the infinite recursion of metaclasses being instances of other metaclasses, the metaclass `Metaclass class` is an instance of `Metaclass`. Finally, in order to define a common behaviour of all classes, the metaclass `Class` is a superclass of all metaclasses, except the metaclasses `Class class`, `Metaclass`, and `Metaclass class` [GR89] (refer to Figure 8.21 for the inheritance and instantiation hierarchy). The class `Behaviour` defines the minimal behaviour for classes, especially their physical representation.

Similar to Smalltalk, each (user-defined) class of CLOS is an instance of the metaobject class `standard-class` or one of its subclasses, and is often referred to as a *class metaobject*. The class of a class metaobject (which is referred to as a *class metaobject class*) is a subclass of the class `standard-object` which in turn is a subclass of the class `t`. The class `standard-object` is the root class of all base-level classes whereas the metaobject class `standard-class` defines the common behaviour of all class metaobjects.

In CLOS, a class is specified using the macro `defclass` which requires i) a class name, ii) a possibly empty list of direct superclasses, and iii) a list of *slot specifications* as arguments. The slot specifications contain information about the names of all slots as well as the corresponding accessors and initialization values.

In general, CLOS implementations divide the execution of so-called *defining forms* (i.e. the macros `defclass`, `defgeneric`, and `defmethod`) and the processing of metaobjects into a three layer structure:

- the *macro-expansion* layer that provides a thin layer of syntactic sugar in order to

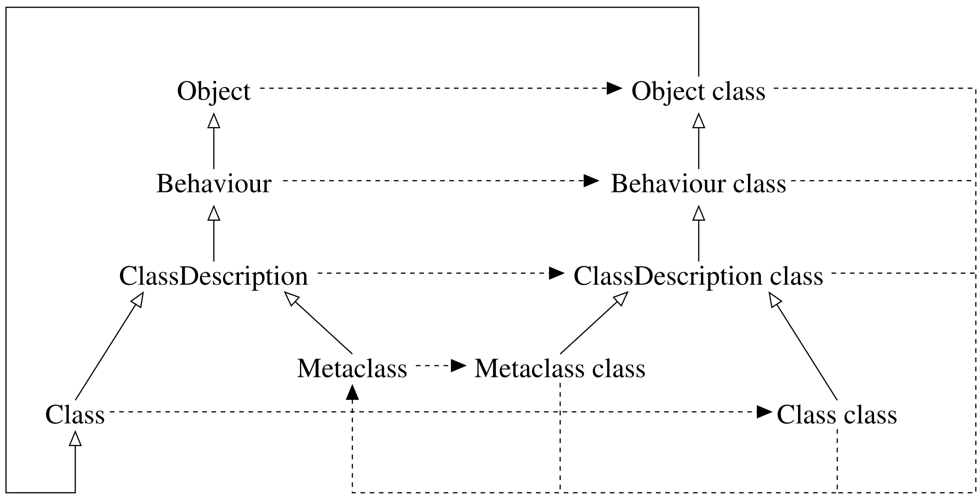


Figure 8.21: Smalltalk class/metaclass kernel.

define classes and methods (i.e. the job of the macro `defclass` is to parse the class definition and to convert it into calls to the glue layer),

- the *glue-layer* that maps names to metaobjects, and
- the lowest layer that defines the behaviour of all standard class metaobject classes.

In Python, classes can be defined using a `class` expression which contains the class name, a possibly empty list of direct superclasses, class variables and methods, but no explicit data elements (i.e. *attributes*) for its instances. At run-time, classes are represented as objects with a specific behaviour. In particular, a class object (like all objects) defines a special attribute `__dict__` where all methods and class attributes are stored and an attribute `__bases__` which contains a (possibly empty) tuple of parent-classes. For all classes which are not defined as subclasses of another class, an empty tuple `()` is specified as the "superclass".

By default, there are no explicit metaclasses. However, as we will point out in section 8.8.6, a restricted form of metaclasses can be introduced in Python using the protocols for class definitions and method lookup.

In the FORM calculus-based object model, classes and mixins are represented as objects, but they are not instances of a metaclass; they are defined by using one of the meta-class level abstractions (which represent encodings of functions), which in turn are created by passing appropriate model generators, model wrappers, and model composers to the abstraction MetaModel. A class abstraction requires the specification of the direct parent-class of a new class as well as an abstraction `Delta` which defines the set of methods which differ in respect to the parent-class. For creating a mixin metaobject, the parent-class is omitted. Unlike Smalltalk or CLOS, instance variables are not explicitly declared, but they are (implicitly) defined in the declaration part of the abstraction `Delta`.

The model generators, model wrappers, and model composers passed to the abstraction MetaModel specify the semantic model of the corresponding meta-class level abstraction. Hence, meta-class level abstractions can be seen as abstractions which are defined by plugging together appropriate functionality for `Create` (i.e. the constructor), `CreateIntermediate` (used for the generation of intermediate objects), and `Compose` (for mixin composition and application): meta-class level abstractions are compositions of appropriate model abstractions.

8.8.2 Structure of instances

In Smalltalk, instance variables specified in a class `C` or any superclass thereof are accessible by name in methods defined on the class `C`. Outside of such methods, only user-defined methods (and low-level implementation loop-holes) can access instance variables. In both CLOS and Python, instance variables are public and are accessible by any client. In the FORM calculus-based object model, instance variables are private and, therefore, can only be accessed by the methods of the class they are defined in. None of the models makes a distinction between public and protected data elements.

Neither Smalltalk[11] nor the FORM calculus-based object model allow an object to change its class at run-time. In CLOS, there is a generic function `change-class` (see section 8.8.3 for details) which can be used to change the class of an instance. Changing the class of an instance creates an new set of initialized slots, but carries over the values of the slots common to both classes. In Python, the class metaobject of an object is stored in an updatable attribute `__class__`. By changing this attribute to another class, the starting point of the method lookup is changed to the new class (see section 8.8.5 for details of method lookup in Python). In contrast to CLOS, however, the instance variables only present in the new class are not initialized.

8.8.3 Operations

In Smalltalk and Python, a method (i.e. an operation on the accessible data elements of an instance) is always associated with the class it is defined in: methods are defined in the context of the class definition. In Python, methods are represented as *function objects* at run-time whereas in Smalltalk, methods are instances of the class `CompiledMethod` (or one of its subclasses). In CLOS, methods are associated as much with so-called *generic functions* as with classes. A generic function definition (using the macro `defgeneric`) specifies the *interface* (i.e. method name, argument list) common to a collection of methods whereas separately defined methods implement the generic function's behaviour for different classes. Both generic functions and methods are represented as metaobjects at

[11] In some Smalltalk implementations, there is a method `changeClassToThatOf` which changes the class of an object. However, both classes must define the same physical structure for their instances [Riv96].

run-time and are instances of the `standard-generic-function` and `standard-method` metaobject classes (ore one of its subclasses), respectively. The metaobject of a generic function has links to all method metaobjects defined on the generic function. A method is associated with a class if a parameter is specialized to that class. When this method is called, the corresponding argument must be an instance of this class (or any of its subclasses). Furthermore, generic functions may be qualified as *before*, *after*, or *around* methods, which we will further discuss in section 8.8.5. In the FORM calculus-based object model, methods are associated with the abstraction `Delta` they are defined in. Although an abstraction `Delta` is always evaluated in the context of a class or mixin metaobject, it can be specified independently of any class and, therefore, can be used more than once as a parameter for defining a class or mixin.

Note that in Python, an additional self parameter has to be explicitly specified for each method whereas self is passed as an argument to the abstraction `Delta` in the FORM calculus-based object model. In Smalltalk, no explicit self is needed. Due to the concepts of generic functions and multiple method dispatch (section 8.8.5), there is no explicit notion of self in CLOS.

8.8.4 Inheritance

Like most object-oriented languages, CLOS, Smalltalk, and Python as well as the FORM calculus-based object model support the specification of new classes as an incremental modification of previously defined classes: they support the notion of inheritance.

The class model of Smalltalk is the only model which does not offer a form of multiple inheritance: only single inheritance is supported. This corresponds to the fact that it is the direct superclass' responsibility to create a new subclass (i.e. sending the message `subclass` to the direct superclass). Furthermore, it is illegal to specify the same name as an instance variable in a subclass when it is already defined in one of its superclasses. Given this restriction, an instance of a subclass contains the union of instance variables named in the class definition and in its superclasses. Any of the inherited methods can be redefined, and an inherited method can be called in the body of its redefinition by invoking the corresponding method on the pseudo-variable `super`.

Python offers support for both single and multiple inheritance. If a method is inherited from more than one superclass, the left-most class in the tuple of superclasses which defines this method has precedence over all other classes. Any inherited method can be redefined, and the number of arguments can be changed, too. In contrast to Smalltalk, Python does not have an equivalent to a `super` call: in the body of its redefinition, an inherited method must be invoked by the appropriately qualified name (e.g. method `foo` of superclass `C` must be called as `C.foo(...)`). This schema ensures that in case of redefining a multiply inherited method, any of the inherited methods can be explicitly called. Due to the fact that data elements are not explicitly specified, an instance of a class also contains the union of instance variables of its superclasses. Data elements with the same name used in more than one (super-)class are shared (i.e. Python supports shared multiple inheritance).

CLOS supports a notion of multiple inheritance similar to mixin-based inheritance discussed in section 8.4. A class inherits both structure and behaviour from all its direct and indirect superclasses in a specified order, from most specific to least specific. The so-called *class precedence list* of a class defines this order by including all of its superclasses in order of decreasing specificity. The class precedence list of a class is computed by calling the generic function `compute-class-precedence-list` and requires that each direct or indirect superclass only appears once. The default algorithm (defined for the class `standard-class`) involves topological sorting the list of direct and indirect superclasses of a class, using local precedence ordering as a constraint [KdRB91]. The full set of slots of a class is the union of the slots of all classes that appear in the class's precedence list. If more than one class in the class precedence list has a slot with a given name, only the the slot definition of the most specific class is retained. Similarly, the set of available methods is specified by the generic functions associated to the classes of the class precedence list. As an equivalent to `super` calls of Smalltalk, CLOS defines the notion of calling the next method (i.e. using `call-next-method`), which will be discussed in the next section.

Both the inheritance model of Smalltalk and Python are defined by the language and cannot be altered. In CLOS, the inheritance behaviour is specified by the class precedence list, and can be changed for different class metaobjects (see section 8.8.6). In the FORM calculus-based object model, the inheritance behaviour is specified by the methods `CreateIntermediate` and `Compose` associated with a class or mixin metaobject. This behaviour depends on the model generator and composer used to create the corresponding meta-class level abstraction. As we have discussed in section 8.6, this allows us to define a much broader range of inheritance mechanisms (e.g. single, shared and repeated multiple, Beta-style) and method dispatch strategies as it is possible with CLOS, Python, or Smalltalk.

In the FORM calculus-based object model, instance variables are private and cannot be accessed by a subclass. However, if two accessor methods (one for read, one for update) are defined on an instance variable, it is possible to overcome this problem. Inherited methods can be redefined by any subclass, and an inherited method can be called in the body of its redefinition by invoking the corresponding method on the intermediate object `orig` passed to `Delta`.

8.8.5 Method dispatch

In contrast to the other class models, CLOS supports so-called *multiple dispatch*: the method to be executed does not only depend on the class of the first argument (i.e. the receiver of the message in Smalltalk terminology), but depends on the classes of all arguments. CLOS divides generic function invocation into three steps: determining which methods are applicable, sorting the applicable methods into decreasing precedence order, and sequencing the execution of the sorted list of applicable methods. Method applicability is decided by looking at the methods specializers: a method is applicable if every required argument satisfies the corresponding parameter specializer. The list of applicable methods is sorted in decreasing precedence order based on the class precedence list of the associated classes (see [KdRB91] for details).

Under the rule of standard method combination, applicable before-methods are executed first, form most specific to least specific. The most specific applicable *primary* method is executed next (a primary method is a method without a method qualifier), followed by the applicable after-methods, from least specific to most specific. If a primary method calls `call-next-method`, the next most specific primary method is invoked.

The reader should note that in the context of inheritance, the "next most specific primary method" for a method is not always the *same* method: depending on the class precedence list, this may vary from class to class. This is of particular importance if a new class is mixed into an existing class hierarchy.

In Python, invoking a method on a object (using `obj.foo(args)`) is nothing else than syntactic sugar for `apply(getattr(obj,"foo"),(obj,)+args)`. The built-in function `getattr` *dynamically* looks up the method with the name of the selector and applies the resulting function object to the list of arguments prepended by the object itself (this explains why methods in Python always require an additional parameter `self`). Note that the function `getattr` is hard-coded in the run-time system and cannot be altered. If the method to be invoked is not defined in the class of an object, the superclasses are searched for this method in a depth-first, left-to-right order. If it is not found in any of the classes, `getattr` invokes the method `__getattr__` with the selector as the argument. By appropriately implementing `__getattr__`, a user can define the required behaviour in case of method lookup failure. This method can also be used as a hook for meta-programming. Furthermore, in the current versions of the run-time system, method lookup is not cached. Therefore, by changing the value of the attribute `__class__`, a method lookup may return a different method than before.

Method invocation in Smalltalk is similar to Python: if a method to be invoked is not defined in the class of an object, then it is recursively looked up in the superclass chain. If none of the classes defines this method, the exception `messageNotUnderstood` is raised, which causes the invocation of the method `doesNotUnderstand` on the receiver of the original message (i.e. the object the method is invoked on). This method can be appropriately overridden in order to define the desired behaviour for method lookup failure. Like in Python, the method lookup functionality is built-in and cannot be altered. Although not specified in the language definition, method lookup is precomputed for efficiency reasons in many Smalltalk implementations (i.e. a method table with "pointers" to all applicable methods for a class is stored in the corresponding class metaobject).

Due to the fact that an object is represented as a form with bindings for all the public methods, method lookup in the FORM calculus-based object model simply consist of a name projection on this form; no rule-based method lookup in a parent-class hierarchy is needed at run-time. Invoking a method consist of sending a form (representing the actual arguments of the method) to the channel bound by the label corresponding to the method name.

From a different point of view, the "method lookup rules" are specified by the generator used for instantiating the class metaobject of an object: a hierarchy of appropriately parameterized intermediate objects is instantiated and composed. This composition "precomputes" the methods to be called on a self- or orig-call. This is one reason why changing the class of an object at run-time would not have an effect on its behaviour.

In contrast to the other models where a lookup failure can be caught an handled, the FORM calculus-based object model does not offer such a feature: projecting an unbound label results in a run-time error.

8.8.6 Adaptability and extensibility

It is generally accepted that the real power of meta-level approaches lies in the fact that the associated metaobject protocols provide mechanisms for adaptation and extension. Each metaobject protocol defines an appropriate default behaviour, and opens the language space into well-defined directions. This implies, however, that a user generally cannot extend or adapt the metaobject protocol arbitrarily, but has to follow a predefined set of restrictions. As the last section of our comparison, we will therefore briefly compare the mechanisms offered for adaptation and extension.

The metaobject protocol of CLOS is the best documented protocol of the three languages we considered, and offers several hooks for adaptation and extension. As we pointed out in this section, the behaviour of all objects, generic functions, and methods of a CLOS application is determined by the behaviour of the class metaobjects used. Therefore, the main mechanism is to subclass one of the existing class metaobject classes and to define new methods specialized to theses classes. As an example, the computation of the class precedence list can be altered by subclassing `standard-class` and specializing the `compute-class-precedence-list` for this new class metaobject class. However, the resulting class precedence list must still fulfill the restriction that each (super-)class only appears once and that `standard-object` and `t` are the last two elements of the list. All the standard defining forms we have mentioned in section 8.8.1 accept an additional parameter specifying the class metaobject class to be used for creating its instances. The mechanism of subclassing class metaobject classes allows one to tailor the change of behaviour of a set of objects only; all other objects are not affected by this change. Discussing the full CLOS MOP is beyond the scope of this work; refer to [KdRB91] for all the details.

Similar to CLOS, the behaviour of Smalltalk can be changed by modifying appropriate methods in classes or metaclasses (changing a method of an existing class will be automatically reflected by all its instances). This mechanism can be used to rearrange the class/metaclass kernel (see Figure 8.21). One of the salient features of Smalltalk is, however, the fully reified compilation process. Since any compiler implicitly gives the semantics of the language it compiles, and because all the Smalltalk implementations have in themselves, as regular objects, their own compiler (including a parser, byte-code generator, scheduler etc.), the semantics of Smalltalk is fully controllable. Therefore, a user may extend the semantics of the language by modifying/extending the available compiler classes. For example, it is possible to redefine the class `MessageNode` which implements the behaviour of message sending.

During the compilation of a new method, the corresponding class is asked which compiler etc. should be used in order to perform the compilation. By overriding the corresponding methods (originally defined in the class `Behaviour`), each metaclass has the possibility to specify the behaviour of its instances. Extending the compiler classes is used in [Riv96] in order to add Eiffel-like pre- and postconditions to the language. Except the subclasses of this class and the class itself, this modification does not change the behaviour of any of the other classes.

The main mechanisms in Python to change the behaviour of classes is i) the hook offered by the method `__getattr__` and ii) by modifying the values of special attributes (e.g. `__dict__` and `__class__`). Furthermore, there exists a (badly documented) way to define class objects as instances of other classes which is based on the notion of *callable objects*. An "ordinary" object is callable (i.e. it can be considered as a function and invoked using the syntax for function calls) if it defines a method `__call__`, whereas a class object must support a method `__init__`. A class object is called whenever a new instance is created. However, if an instance of a "metaclass" is subclassed, the metaclass is called in order to instantiate the new class object appropriately. Unfortunately, the sequence of function calls and the corresponding parameters is hard-coded into the Python run-time system, and is not documented in the language reference.

The simple protocol of the FORM calculus-based meta-level framework is mainly based on the behaviour of the different model generators, model wrappers, and model composers passed to the abstraction MetaModel. Method dispatch and incremental derivation, for example, depend on how intermediate objects are composed and where fixed-point operators are applied. Hence, by plugging together appropriate model abstractions, the behaviour of the resulting meta-class level abstractions can be tailored to specific requirements. However, since the meta-level framework is built "on-top" of the core of PICCOLA(F), it is not restricted to these mechanisms; any abstraction which returns an object-like structure (i.e. a form representing the interface to a set of interconnected agents) can be considered as a class abstraction.

Unlike in Smalltalk, the run-time system of PICCOLA(F) is not reified and therefore, it is not possible to have such a fine-grained control over the behaviour of applications.

8.9 Related work

Over the last decade, several researchers have proposed foundational models for object-oriented programming. Although most of these models are strongly based on typed λ-calculi with subtyping, stylistic differences make a rigorous comparison difficult. Some models, for example, are presented as translations from high-level object syntax into the syntax of a typed λ-calculus whereas others map high-level syntax directly into a denotational model or focus on the object syntax as a primitive calculus in its own right. Common to all models is, however, that due to the usage of the λ-calculus as a formal framework, they neither address concurrency nor distribution.

Examples of such foundational models are the recursive-record encodings of Cardelli [Car88], Reddy [Red88], and Cook [Coo89], existential encodings proposed by Pierce and Turner [PT94], Bruce's model based on existential and recursive types [Bru94], and the type-theoretic encoding of a calculus of primitive objects defined by Abadi et al. [ACV96]. Further work in this area includes a calculus for delegation-based languages by Fisher and Mitchell [FM95] and a calculus of classes and mixins proposed by Bono et al. [BPS99]. A detailed comparison of all these λ-calculus based approaches is beyond the scope of this work (refer to [AC96] or [BCP97] for a further discussions).

While formal object models based on the λ-calculus emphasize aspects like encapsulation, classes, inheritance, and incremental modification (e.g. method update), models that address concurrency focus on aspects such as concurrency, active objects, distribution, and synchronization. Most of the concurrent models are based on some process calculus (CSP, CCS, π-calculus, join-calculus [FG96]) or on (asynchronous) actor models [Ahg86]. For a detailed survey of formal models for object-orientation that address concurrency, refer to [Men94].

Papathomas, for example, has defined a framework for describing the semantics of concurrent object-based programming languages in CCS [Pap92]. This framework captures some essential concepts of these languages (e.g. concurrent objects, classes, inheritance), but neither takes into account classes as first-class objects and nor offers support for delegation- and prototype-based languages. Nierstrasz has proposed OC, a process-based calculus designed to provide a formal semantics for concurrent object-based programming languages [Nie92]. This calculus integrates the concepts of both agents and functions and tries to capture the three fundamental aspects of concurrent object-based programming: encapsulation, active objects, and composition.

Variants of the π-calculus have been previously used by other researchers to model various aspects of object-oriented programming languages. Walker has shown that POOL [Ame87] can be modelled in the π-calculus, but in his approach, no subtyping or inheritance is supported [Wal95]. The same applies to the encoding of the object-oriented design notation $\pi o \beta \lambda$ (pronounced "pobble") in the polyadic π-calculus proposed by Jones [Jon93], and the translation of the object model by Abadi et al. [ACV96] into the π-calculus presented by Sangiorgi [San96b]. Subtyping and a notion of self can be modelled with the "Calculus of Objects" of Vasconcelos [Vas94]. Barrio has given a nearly complete representation of active objects in the π-calculus, but both dynamic binding and a notion of self are still missing [BS95].

The concept of mixins has been proposed by several researchers in order to overcome some of the problems with multiple inheritance [Coo89, BC90]. Van Limberghen and Mens give a denotational semantics of their mixin model, where mixins, mixin composition, and encapsulation are primitives, but they do not incorporate an explicit notion of classes [VLM96]. All these concepts are integrated as primitives in the calculus of classes and mixins proposed by Bono et al. [BPS99]. Ancona and Zucca have studied a rigorous semantic foundation for mixins independently from the notions of classes and objects, starting from an algebraic setting for module composition [AZ96].

A slightly different approach is used by Flatt et al. where a subset of Java is extended with mixins [FKF98]. Their system supports higher-order mixin composition, a hierarchy of named interface types, and resolution of accidental name collisions. In contrast to explicit encapsulation, the collision resolution system allows the "original" and the overriding method definitions to coexist. The two methods are distinguished using so-called *views* on objects, which is carried with the object at run-time and altered at each subsumption step. As a consequence, method lookup is sensitive to an object's history of subsumption [FKF98].

In order to model the mechanism for inheritance of several object-oriented languages, in particular multiple inheritance of C++, Rossie et. al define the notion of inheritance in terms of so-called *subobjects* [RFW96]. From their point of view, a class `C` represents a collection of members (i.e. the methods and instance variables that are shared by all instances of `C`). When a class `D` inherits from `C`, the underlying inheritance mechanism may either attempt to merge the members of `C` with those of `D` or collapse members with the same name into a single definition. Alternatively, all members of `C` are inherited as an indivisible collection. This collection, when instantiated, is known as a subobject. Each instance of the class `D` has a distinct subobject `D/C` as well as a subobject `D/D`; the latter is also referred to as the *primary subobject* of `D`. Subobjects are meant to support subclass polymorphism: each subobject represents a different view of an object, allowing it to be viewed as an instance of any of its parent-classes.

In this model, an instance cannot be seen as a simple record-like structure with the member names as field names: only the subobjects are represented as records whereas an instance has to be considered as a *collection of subobjects*. Member references are made by i) selecting the appropriate subobject that defines this member and ii) by referencing the corresponding field of that subobject. The inheritance model of a particular language specifies which subobject has to be selected in the context of self- and/or super-calls, respectively.

8.10 Summary

Throughout this chapter, we have defined a basic object model in the FORM calculus (section 8.1) and several extensions to this model (sections 8.2 to 8.5). This resulted in the specification of a meta-level framework for concurrent, object-oriented programming abstractions (section 8.6) which generalizes approaches presented by Bracha and Cook [BC90], Cook and Palsberg [CP94], Van Limberghen and Mens [VLM96], and Rossie et. al [RFW96]. We also pointed out situations where the expressive power of the FORM calculus is not enough and expressions beyond the calculus are needed (section 8.7), and compared our object model with other meta-level models (section 8.8). In this section, we briefly summarize the main observations of our modellings and the meta-level framework.

The essentials of concurrent objects in the FORM calculus are captured by our meta-level framework: an object is viewed as an agent containing a set of local agents and channels representing methods and instance variables, respectively, whereas the interface of an object is a form containing bindings for the channels of all exported features. Furthermore, the framework defines an imperative object model with an early bing of self [BPS99], data encapsulation, and low-level synchronization.

Representing classes as first-class run-time entities (i.e. *class metaobjects*) allows us to integrate features of object-oriented programming such as class variables and methods, various inheritance mechanisms, different method dispatch strategies, and higher-level synchronization abstractions. The need to use class metaobjects arises naturally when we want to model a correct initialization of and a controlled access to class variables and methods. Various inheritance mechanisms are achieved by introducing *intermediate objects* (i.e. objects with an unbound self-reference) specifying partial behaviour of objects, *generator agents* defining compositions of intermediate objects, and *wrappers*, which apply a fixed-point operator over composed intermediate objects to establish a sound interpretation of self. Different method dispatch strategies are modelled by applying a fixed-point operator at various stages of intermediate object composition. From a different point of view, intermediate objects can be considered as *subobjects* [Red88] with an unbound self-reference whereas a generator specifies the method lookup and dispatch rules of the corresponding inheritance model. Note that the mechanism of polymorphic form extension defined in the FORM calculus is an essential feature in order to define the composition of intermediate objects in a generic way.

Unlike many object-oriented programming languages which introduce several levels of visibility for attributes, the FORM calculus-based object model only makes the distinction between *private* and *public* attributes. It ensures that private attributes (instance variables and methods) are only visible within the scope of the abstraction `Delta` of a given metaobject. As a consequence, private attributes can only be accessed in the class they are defined in, but not in any of its subclasses. This implies that a subclass can define a new (private or public) attribute with the same name without interfering with a private attribute of one of its parent-classes. Furthermore, our model ensures that instance variables are correctly initialized: this is achieved by appropriately parameterized abstractions `Delta` and by sequencing the composition of intermediate objects. Note that correct initialization of instance variables heavily depends on keyword-based parameter passing.

A generalization of the concepts of generators, wrappers, and composition of intermediate objects cannot only be used to define classes and class abstractions, but also to model mixins, mixin application, mixin composition as well as method encapsulation. By i) splitting the functionality of generators and wrappers into a static protocol-part and a variable model-part and ii) introducing the concept of a *composer abstraction*, it is possible to derive all object-oriented abstractions mentioned above from a single meta-model abstraction MetaModel. Whereas this meta-model abstraction defines the generic behaviour of all meta-class abstractions and the corresponding meta-protocol, so-called *model* generators, wrappers, and composers specify the common behaviour for specific semantic models. This approach also allows us to separate the semantic model of concrete classes and mixins from their specific behaviour.

The concept of a composer abstraction enables to mixin meta-level behaviour and can, for example, be used to define singleton behaviour as a mixin, which can be applied to any existing class. This approach is in contrast to other meta-level approaches where a class with singleton behaviour needs to be an instance of a singleton metaclass.

Analyzing the mechanisms used to define the meta-level framework, we can see that forms, keyword-based parameters, polymorphic extension, and polymorphic restriction are the key concepts for the resulting extensibility, flexibility, and robustness. In particular, these concepts allows us to make a clear separation between functional elements (i.e. methods) and their compositions (i.e. inheritance), enhance the definition of various semantic models supporting different kinds of inheritance and method dispatch strategies, and clarify concepts which are typically merged in existing programming languages. Hence, these concepts enable the definition of a canonical set of features for concurrent, object-oriented programming based on a small set of primitives, and any meta-class level abstraction can be seen as a composition of such features. This approach allows us to define multiple class models using a single semantic framework and, therefore, substantially enhances the possibilities to bridge compositional mismatches in heterogeneous applications. Furthermore, our approach overcomes problems of related approaches which integrate concepts in a non-orthogonal way, and define higher-level features as primitives.

The reader should note that presented meta-level framework can also be defined in a (non-concurrent) environment that provides the same expressive power for record-like structures as the FORM calculus. In particular, keyword-based parameters as well as asymmetric record concatenation and restriction must be available.

Chapter 9

Compositional abstractions

In the previous two chapters, we have defined the FORM calculus as a formal foundation (chapter 7) and used this foundation for defining a meta-level framework for concurrent, object-oriented programming (chapter 8). However, it is not our goal to focus only on the definition of object-oriented programming abstractions, but to use the FORM calculus as a formal foundation for any kind of compositional abstraction. Therefore, we will specify a FORM calculus-based component framework for stream and filter composition to exemplify our view of composition, and analyze various approaches to define the corresponding connectors in an flexible and extensible way.

In section 7.2, we illustrated the expressive power of polymorphic form extension in the context of extensible, higher-level composition abstractions, and showed that polymorphic extension enables the composition of arbitrary services in a generic way. In this section, we go a step further and illustrate that keyword-based parameters, polymorphic form extension and restriction in combination with matching are the key concepts for defining compositional abstractions in a flexible and extensible way: none of the discussed abstractions can be easily expressed in a generic and robust way without using the concepts mentioned above. Furthermore, we argue that a component framework that incorporates an operator-based approach for implementing its connectors may substantially benefit from the expressive power of the key concepts in order to enhance the separation between computational elements and their relationships.

This chapter is organized as follows: in section section 9.1, we define a component framework for stream and filter composition (similar to the Bourne Shell). We continue with a comparison of both dispatched-based and operator-based approaches for defining the connectors of the filter framework. We conclude this chapter with a summary of the main observations. Note that we will again use PICCOLA(F) as an executable specification language in order to illustrate our encodings at a higher level of abstraction.

9.1 A framework for stream and filter composition

In section 2.2, we defined a software component as a static abstraction with plugs and as a composable element of a component framework. Furthermore, we defined a component framework as a collection of software components with a software architecture that determines the interfaces that components may have and the rules governing their composition. Hence, by definition, any framework which adopts this view can be considered as a component framework.

We also pointed out that a component framework can be viewed as a many-sorted algebra where the components are the operands and the corresponding composition mechanisms are the operators (refer to section 2.4). In this section, we define a component framework for stream and filter composition (similar to the Bourne Shell) which exemplifies our algebraic view components, frameworks, and composition. However, it is not the goal of this chapter to focus on the components of the framework; we mainly concentrate on i) how the *connectors* of the framework can be defined and implemented and ii) what kind of *concepts* and *mechanisms* are needed to do so. In order to simplify the following discussion, we assume that the interface of a component is represented as a form.

Similar to the Bourne Shell [Bou78], the component framework for stream and filter composition consist of i) *data sources* (denoted by the character 'Q'), ii) *data sinks* (denoted by 'S'), and iii) *filters* which read from an input-stream and write to an output- and/or error-stream. Note that a filter does some processing of the data read from its input-stream and produces output onto its output- and/or error-stream, but for the following discussion, the actual data processing is not important, and has therefore been omitted.

For the rest of this chapter, we denote a filter as an element of the set {IOE, IO, IE, OE, I, O, E, ∅}. In fact, the set {IOE, IO, IE, OE, I, O, E, ∅} denotes the *sorts* of the corresponding algebra. The presence of a character 'I', 'E', or 'O' indicates that the corresponding I/O-stream is unbound in a filter (e.g. 'IOE' denotes a filter which has all of its I/O-streams still unbound whereas 'O' denotes a filter where only the output-stream is unbound). Since all I/O-streams are bound, the filter '∅' cannot be used for further composition. This is in contrast to Bourne Shell scripts where '`cat infile | sort >& outfile`' cannot interact with any other Unix filter (all I/O-streams are connected), but can be used as a component in another pipe and filter chain.

The connectors of the framework are defined by the set $\{<, |, >, |\&, > \&\}$. Each of the connectors creates a new stream in order to connect a pair of unbound I/O-streams of either two filters, a data source and a filter, or a filter and a sink. The connectors are defined in a way that applying a connector to two components leads to a new component which may be used for further composition.

Like in the Bourne Shell, a pipe operator '$|$' can be used to connect an unbound output-stream of a filter component with the unbound input-stream of another filter component (e.g. composing two filter components '$\mathrm{I}_1\mathrm{O}_1$' and '$\mathrm{I}_2\mathrm{O}_2\mathrm{E}_2$' leads to a composite filter component[1] where 'O_1' is connected with 'I_2', but all other I/O-streams are left unbound: it yields '$\mathrm{I}_1\mathrm{O}_2\mathrm{E}_2$'). If both filters composed with the '$|$' operator have an unbound error-stream, the two error-streams are merged (i.e. the resulting component must ensure that if its error-stream is connected, the error-streams of the two filters are also bound to this

[1]A composite filter component should not be confused with the *composite pattern* discussed in [GHJV95].

stream). The other connectors are specified similarly and ensure that i) no feedback loops can be introduced between components and that ii) a connector can only be applied to two plug-compatible components (e.g. a pipe operator '|' can only be used in a context where the left-hand side component has an unbound output-stream and the right-hand side component an unbound input-stream). For the complete specification of the sorts of the corresponding algebra and how the operators map components of the various sorts to other components, refer to appendix D.

For the rest of this chapter, we will discuss several possibilities how the connectors '$<$', '$|$', and '$|\&$' can be defined and what kind of abstractions are needed to implement them. The remaining operators are similar, and a discussion has therefore been omitted.

9.2 Dispatch-based approaches

As a first approach for implementing the connectors of the filter framework, we assume that a component is represented as an object-like structure: it is able to execute method invocations. This allows us to define the composition of two components as a method call on the left-hand side component (i.e. '$F < Q$' is translated into `F.indirect(Q)`, '$F_1 | F_2$' into F_1`.pipe(`F_2`)`, and '$F_1 |\& F_2$' into F_1`.pipeAmp(`F_2`))`). This approach has the consequence that the connectors are associated to the components and, therefore, the correct implementation of the connectors is the responsibility of the components themselves.

9.2.1 Single dispatch

As we can deduce from the specification of the operator '$<$', the sort of the resulting component (i.e. the set of the unbound I/O-streams) only depends on the sort of the left-hand side filter, but not on the sort of the right-hand side data source. Therefore, it is straightforward to implement the operator '$<$' using a single-dispatch strategy: any filter component defines a service `indirect` (taking a data source as an argument) which binds its input-stream to the data source and returns a component representing the composition of itself and the data source.

In order to reflect the fact that the input-stream is bound, the resulting component should not offer services which connect its input-stream with another stream. Therefore, the resulting component can be defined as a wrapper around the filter and the data source which *restricts* access to some of the services (e.g. all services which connect the components input-stream), but forwards all other service invocations to the filter and the data source, respectively.

In a pure object-oriented programming language (such as Eiffel or Java), an object is always an instance of a class. Therefore, it is necessary to add a set of new classes to the framework which define the behaviour of composite components. If the underlying language offers hooks for method lookup (such as Python or Smalltalk), it is often enough to define a single composite class which acts as a wrapper around two objects and forwards method calls to the appropriate object (e.g. the class `CompositeStream` illustrated in appendix A.1 uses this mechanism). This also allows us to define a generic method `indirect` which simply passes **self** and the right-hand side argument to the constructor of the composite class.

In Eiffel or C++, where such meta-level hooks are not available, it is necessary to define a set of new classes representing composite components (e.g. four composite classes for filter and data source composition). Furthermore, each filter component must know which of the composite classes must be instantiated: this can either be implemented by overriding the method `indirect` for each filter class or by defining `indirect` as a *template method* [DW99].

Using the FORM calculus-based object model, where an object-like structure is not necessarily an instance of a class metaobject, it is possible to omit an abstraction for a composite class: the resulting component can simply be expressed using binding restriction:

```
function indirect (Args) =      {- defined for each filter class -}
     self().In (<Args.source()>);
     < self() \ In \ indirect >
```

In order to simplify the example, we assume that a data source (given by `Args.source()`) offers the same interface as a stream component and can be directly connected to the input-stream of a filter by calling the method `In`. The method `indirect` substantially benefits from keyword-based parameters as it only requires that the component itself offers a method `In`; no further information is required. Finally, restricting the resulting composite component with `In` and `indirect` ensures that it cannot be used in contexts where an unbound input-stream is required.

Note that it would also be possible to define `indirect` without binding restriction, but this would imply that i) *all* services of the corresponding component have to be known and that ii) `indirect` is not generic enough for future extensions (refer to the discussion about the abstractions $\mathit{fixcomp}$ and $\mathit{compose}$ in section 7.2).

Without using some kind of introspection mechanism, it is not possible to define the two pipe operators '$|$' and '$|\&$' using a single dispatch approach: the resulting component depends on the sorts of both filter components to be composed. Possible implementations are very similar to the ones presented in section 9.3, and have therefore been omitted here.

9.2.2 Double dispatch

As mentioned above, the result of an application of the pipe operator '$|$' depends on the sorts of both filter components to be composed. This situation is very similar to the one we encountered for modelling mixin composition (refer to section 8.4.2), and it is possible to use an approach based on a *double dispatch strategy*: the method `pipe` of the left-hand side filter component sends information about itself to the right-hand side filter by calling an appropriate method.

As we can see in the specification of the pipe operator '|' in appendix D, there are two situations we have to distinguish: the first situation where both filters to be composed have an unbound error stream (which need to be merged), and the second situation where at most one of the filters has an unbound error stream. Therefore, a filter component must define two methods `connectErr` and `connect` which reflect these two situations and define the correct composition of two filter components. However, as we illustrate below, we need two more methods `connectIn` and `connectInErr` in order to pass on the information about the status (bound or unbound) of the input-stream as well.

Modelling the pipe operator '|' in the FORM calculus is not as straightforward as implementing the operator '<'. For each possible combination of unbound input-, output-, and error-streams, a corresponding composite class has to be defined (i.e. eight composite classes for the filter framework). This is necessary to ensure the correct behaviour of a composite component in the context of further composition.

In order to explain the problem, reconsider the situation of composing the filters $F_1 = I_1O_1$ and $F_2 = I_2O_2E_2$: as mentioned above, '$F_1 \mid F_2$' is translated into F_1`.pipe(`F_2`)`. Due to the fact that the filter F_1 has an already bound error-stream, it calls F_2`.connectIn` with `self` as argument. Invoking the method `connectIn` tells the filter F_2 that the argument has an unbound input-stream, but an already bound error-stream and, therefore, no merging of error-streams is required.

As a first approach, the method `connectIn` of a filter could be defined as follows (analogous to the method `indirect` discussed in the previous section):

```
function connectIn (other) =      {- defined for each filter class -}
    let
        value str = Stream.Create()
    in
        other.Out (<str>);        {- binding of 'Out' and 'In' -}
        self().In (<str>);
        <
            other \ Out \ pipe \ indirect,
            self() \ In \ connect \ ... \ connectInErr
        >
    end
```

The method `connectIn` creates a new stream, binds this stream to the unbound input- and output-stream of the two filters, and returns a composition of the two filters where the appropriate services have been removed (e.g. `Out`, `pipe`, and `indirect` are removed from the left-hand side filter since the output-stream is bound and, therefore, it cannot be used as the right-hand side argument of a filter composition or data source redirection).

At a first glance, this seems to be the correct way to do filter composition. However, a problem arises when the filter $F_3 = I_3O_3E_3$ is composed with $F_{12} = F_1 \mid F_2$ (i.e. $F_{3,12} = F_3 \mid F_{12}$): the filter F_3 has unbound input- and error-streams and, therefore, it invokes the method `connectInErr` on the composite filter F_{12}. Due to the way F_1 and F_2 are composed, the `connectInErr` of F_1 is called, which defines the behaviour for a filter with an already bound error-stream. This, however, is not the case for the composite filter F_{12}, and the resulting composite filter $F_{3,12}$ has a dangling error-stream E_3.

In order to solve the problem, all four connecting methods (`connect`, `connectIn`, `connectErr`, and `connectInErr`) must internally instantiate an instance of a corresponding composite filter class (instead of simply using binding restriction and polymorphic form extension). Unfortunately, none of the composite filter classes can benefit from polymorphic form extension in order to enhance their extensibility.

The main problem with a double-dispatch approach is that the method `pipe` must send information about itself to the appropriate connect method of the right-hand side filter component. However, using binding restriction and polymorphic form extension as illustrated in `connectIn` above, the information that two filter components F_1 and F_2 are composed and form a composite component is neither propagated to F_1 nor to F_2. This has the consequence that the value of self used in `connectIn` of the composite component still reflects the original value of F_1, and not of the composite component F_{12}. The same applies to any of the other connect methods as well as for the method `pipe`.

The operator '$|\&$' merges the output- and error-stream of the left-hand side filter (if both are still unbound) and connects the merged streams to the input-stream of the right-hand side filter. If one of the two output-streams is already bound, only the remaining unbound stream is connected. Like for the pipe operator '$|$', it is again possible to use an approach based on double dispatch: the method `pipeAmp` sends information about itself to the right-hand side filter by calling an appropriate connect method. If this is done in a straightforward way, eight methods have to be defined for each filter class in order to correctly communicate the required information about the status of all I/O-streams of the receiver of the method `pipeAmp`.

Due to the fact that merging of streams only depends on the status of the output- and error-stream of the left-hand side filter component, the number of connecting methods can be reduced to two. In fact, it is even possible to use two of the connect methods defined for the pipe operator '$|$'. This can be achieved by merging, if necessary, the output- and error-stream in the method `pipeAmp` and passing on only the merged stream to be connected. As an example, consider the method `pipeAmp` for a filter where all I/O-streams are still unbound (i.e. a filter of the sort 'IOE'):

```
function pipeAmp (other) =
    let
        value merged = <
            Out = streamMerge(<s1=self().Out, s2=self().Err>).merge >
    in
        other.connectIn (< self() \ Err, merged >)
    end
```

The method `pipeAmp` merges the two unbound output- and error-streams (refer to Figure 9.1 for details of the abstraction `streamMerge`) and dynamically extends the filter component with the resulting stream, accessible by the service `Out` (i.e. the filter component pretends that only an unbound output-stream has to be connected). More precisely, the left-hand side filter component does not send itself, but a composition of itself and the merged output- and error-streams to the right-hand side filter component. Hence, in this context, the abstraction `streamMerge` is used as a generic *glue abstraction*. If a left-hand side filter has no unbound input-stream, the method `connect` instead of the method `connectIn` is called. It can be easily verified that this approach results in the correct behaviour of the resulting composite component. The reader should note that the approach used above can in general only be used in an object-oriented programming language if an additional composite class (representing a filter component with merged output- and error-streams) is introduced.

9.2.3 Multiple dispatch, method overloading

In object-oriented programming languages that support multiple method dispatch (such as CLOS), it is possible to omit the connecting methods, define a generic method `pipe`, and implement a specialization for each possible filter class as the second argument.[2] However, due to the fact that the resulting component of each specialization depends i) on the status of the input- and error-stream of the receiver (i.e. the left-hand side argument of the pipe operator) and ii) on the status of the output- and error-stream of the second argument, sixteen different specializations of the method `pipe` have to be defined. Like in the double-dispatch approach discussed in the previous section, all methods `pipe` explicitly specify the result as being an instance of one of the eight composite filter classes.

Another approach for defining the connectors of the filter framework is based on *method overloading* [CW85]. In contrast to multiple method dispatch of CLOS, where the method selection is based on dynamic or *run-time* information, method overloading (such as in C++ or Java) is based on static or *compile-time* information (i.e. the static type of objects). However, the problems discussed for multiple method dispatch also occur for method overloading and, therefore, neither the number of specializations for the overloaded method `pipe` nor the number of composite classes can be reduced. Furthermore, method overloading requires more care for defining compositions as the usage of polymorphic variables may lead to an incorrect behaviour (i.e. the wrong overloaded method is selected based on the static type of a polymorphic variable).

9.3 Operator-based approaches

In contrast to the dispatch-based approaches discussed in the previous section, where the connectors are associated to the components, we illustrate an approach in this section where the connectors are defined *independently* of the components. More precisely, ‘F $<$ Q’ is translated into `indirect(P,Q)` whereas ‘$F_1 \mid F_2$’ and ‘$F_1 \mid\& F_2$’ are translated into `pipe`(F_1,F_2) and `pipeAmp`(F_1,F_2), respectively. Furthermore, we assume that components only offer services `In`, `Out`, and `Err` to connect the corresponding I/O-streams.

As discussed in section 9.2.1, the sort of the composite component of a data source and a filter composition only depends on the sort of the filter and can be expressed by

[2]We consider the first argument of a multi-method as the receiver.

using binding restriction. The abstraction defined below is a simple adaptation of the corresponding function `indirect` (the abstraction for the operator '>' follows an almost identical scheme). Note that the method `indirect` again benefits from keyword-based parameters:

```
function indirect(Args) =      {- defined outside filter classes -}
     Args.filter().In(<Args.source()>);
     < Args.filter() \ In >
```

The correct composition of two filter components using either the operator '|' or '|&' depends on the sort of both components. The corresponding implementation of the two operators have to find out whether two unbound streams have to be merged: two error streams in case of the operator '|', an output- and an error-stream in case of the operator '|&'. Unless a dispatched-based approach is used, some form of run-time introspection is required to retrieve this information.

The reader may have noticed that the presence of an service `In` indicates that the input-stream of a component is still unbound, and that the absence of `In` reflects an already bound input-stream. The same schema also applies for the output- and error-streams. In order to check the status of an I/O-stream of a component (e.g. the presence or absence of a service `In`), it is possible to use the matching operator defined in the FORM calculus: the syntax `[F <- l]` is used in PICCOLA(F) for matching and corresponds to $[F \leftarrow l]$.

In order to implement the abstraction `pipe`, we use an approach based on matching, polymorphic extension, and restriction (refer to Figure 9.1 for the full source code). Similar to the dispatch-based approach, the abstraction `pipe` creates a new stream, connects the output-stream of the left-hand side component with the input-stream of the right-hand side component, and restricts access to the corresponding services `In` and `Out`. Then it checks whether both arguments have an unbound error-stream. If this is the case, the two error-streams are merged (using the abstraction `streamMerge`), and the resulting composite component is expressed as a polymorphic extension of the forms representing the interfaces of both components and the merged error-streams. Note that it is not necessary to restrict access to `Err` of both arguments to `pipe` as the binding for `Err` is overridden by the merging of the error-streams.

If at most one of the components has an unbound error stream, no streams have to be merged, and the resulting composite component can simply be expressed as a polymorphic extension of the forms representing both components, restricted by the corresponding services `In` and `Out`, respectively.[3]

The abstraction `pipeAmp` (which implements the operator '|&') uses a similar approach: it checks whether the left-hand side component has both an unbound output- and error-stream, merges these two streams (if necessary), and returns the resulting component as a polymorphic extension of the forms representing both filter components, restricted

[3]The code of `pipe` in Figure 9.1 could be written in a more compact way using the concept of an *early return*. However, this concept is not yet fully understood in the context of encoding functions in the FORM calculus, and has therefore not been implemented in PICCOLA(F).

```
function streamMerge(Args) =
    <
      function merge(Stream) = Args.s1 (Stream); Args.s2 (Stream)
    >

function pipe(Args) =                    {- operator '|' -}
    let
      value str = Stream.Create()
      value rf1 = Args.f1() \ Out
      value rf2 = Args.f2() \ In
    in
      Args.f1().Out(<str>);              {- binding of 'Out' and 'In' -}
      Args.f2().In(<str>);
      if [rf1 <- Err] then
        if [rf2 <- Err] then
          <
            rf1,
            rf2,
            Err = streamMerge(<s1 = rf1.Err, s2 = rf2.Err>).merge
          >
        else
          < rf1, rf2 >
        end
      else
        < rf1, rf2 >
      end
    end

function pipeAmp(Args) =                 {- operator '|&' -}
    let
      value str = Stream.Create()
    in
      Args.f2().In(<str>);
      if [Args.f1() <- Err] then
        if [Args.f1() <- Out] then
          streamMerge(<s1=Args.f1().Out, s2=Args.f1().Err>).merge(<str>)
        else
          Args.f1().Err(<str>)
        end
      else
        Args.f1().Out(<str>)
      end;
      < Args.f1() \ Out \ Err, Args.f2() \ In >
    end
```

Figure 9.1: Source code of filter framework composition operators.

by the corresponding services `In`, `Out`, and `Err`. Note that, in contrast to the abstraction `pipe`, it is possible to define the resulting composite component by using a single expression.

The illustration of the composition operators defined above shows that the approach of independently defined connector abstractions has the advantage that no composite classes have to be defined. The composite components can be expressed using polymorphic form extension and restriction whereas the required run-time information can be retrieved by using the matching abstraction of the underlying FORM calculus. The resulting composition operators are much more generic and robust in comparison to the dispatch-based approaches as they benefit from the extensibility of polymorphic form extension. As an example, consider the case where we extend the framework with i) a new sort of filter component which can handle an additional input stream (denoted by 'A') and ii) a corresponding set of new connectors. Existing connectors defined using an operator-based approach are not affected by this extension (they do not depend on the presence or absence of an unbound 'A' stream) and can be reused as is. However, connectors defined using a dispatch-based approach are affected by the extension and have to be reimplemented accordingly.

Furthermore, the self problem of the dispatch-based approach (i.e. self of a part component does not reflect the composite component it is part of) does not occur in an operator-based approach.

Note that the operator-based approach illustrated above can only be applied in an object-oriented programming language if it i) offers some kind of introspection mechanisms, ii) allows a user to define abstractions independent of classes, and iii) objects can be expressed as "compositions" of other objects. Object-oriented programming languages such as C++, Java, Eiffel, or Smalltalk lack at least one of these features.

Similar to method overloading discussed in section 9.2.3, *operator overloading* can be used to define the composition operators of the filter framework. Without using any form of run-time introspection, this would lead to four different overloaded methods for both pipe operators '$|$' and '$|\&$' (provided a programming language supports object composition). In languages such as C++ or Java, which do not support object composition, it is again necessary to define a set of composite classes, and the number of overloaded operators increases accordingly.

9.4 Discussion

In the following, we will briefly discuss the main observations of using the FORM calculus for modelling compositional abstractions. Although the stream and filter component framework we used in this chapter is a simplified example, it nevertheless shows important properties of concepts found in the FORM calculus, and reveals the expressive power of keyword-based parameters and polymorphic form extension in combination with polymorphic restriction and matching.

Since the interfaces of components are represented as forms, polymorphic form extension and restriction are the basic mechanism for defining connectors and, by definition, support an algebraic view of composition. In particular, if a component C_1 has a required service or port P_1 and a component C_2 a provided service P_2, then the composition of C_1 and C_2 can be most naturally defined by connecting the two ports and by expressing the resulting component as $<\mathrm{C}_1 \setminus \mathrm{P}_1,\ \mathrm{C}_2 \setminus \mathrm{P}_2>$. The composition of C_1 and C_2 could also be defined by simply using projection and binding extension, but this requires knowledge about the *full interface* of both C_1 and C_2 and, therefore, restricts the reusability of the corresponding connector. The main advantage of polymorphic form extension in combination with restriction is that less restrictions on the interfaces of the components to be composed have to be made, and that connectors can be expressed in a more generic and extensible, hence reusable way.

Note that the composition of C_1 and C_2 cannot be expressed using polymorphic form extension alone as the resulting composite component would still offer a service P_1, which is probably not the intention behind the composition of C_1 and C_2. The usage of restriction allows a user to explicitly select the required service(s) in case of name collisions whereas polymorphic form extension always gives precedence to the right-most service(s). Furthermore, a combination of polymorphic extension and restriction can be used to dynamically adapt the interface of a component (refer to the operator `pipeAmp` defined in the previous section). This was one of the main reasons why polymorphic form restriction was introduced in the FORM calculus.

Although the combination of polymorphic form extension and restriction offers a powerful tool for defining compositional abstractions, there are situations where a composition framework must offer some sort of introspection mechanism in order to inspect the interface and capabilities of components. As we have shown in section 9.3, the decision of how to compose a component with other components may depend on the presence or absence of a given set of services. This is one of the reasons why the matching mechanism defined by the FORM calculus (which can be used as a simple abstraction to inspect the interface of a component) offers a continuation for both a positive and a negative match, respectively.

The connectors of a component framework can either be defined using a dispatch-based or an operator-based approach. The advantage of a dispatch-based approach is that run-time introspection can in general be avoided (e.g. by using a double dispatch strategy). However, as we have shown in section 9.2, such an approach may lead to a set of additional composite component abstractions which ensure the correct behaviour of a composite component in the context of further composition. This often implies that the corresponding connector implementations must explicitly define the resulting composition as an instance of composite component abstraction, which may lead to reimplementations in the context of a framework extension. Furthermore, adding component abstractions to the framework results in a steeper learning curve before the framework can be successfully used.

Operator-based approaches do in general not suffer from the problems of dispatch-based approaches and do not require additional composite component abstractions. Furthermore, as we have shown in section 9.3, they can be defined in a much more extensible and robust, hence reusable way. Based on these observations, we argue that a component framework that incorporates an operator-based approach for implementing its connectors can be adapted to new requirements much easier than a framework using a dispatch-based approach, in particular as an operator-based approach enhances the separation between computational elements and their relationships.

Although the example framework used in this chapter is neither exhaustive nor canonical, it exemplifies an algebraic view of component frameworks, and we argue that the results represent to a certain degree important concepts for component-based software development in general and for defining compositional abstractions in particular. In fact, the results show that the concepts of keyword-based parameters, polymorphic form extension and restriction in combination with matching form a powerful tool for software composition, as these concepts offer the required expressive power to define higher-level compositional abstractions in a flexible, extensible, and robust way. Nevertheless, additional experiments with different kinds of component frameworks are needed in order to further justify this claim.

Chapter 10

Summary of observations

In Part III of this thesis, we have defined the FORM calculus, a conservative extension of the π-calculus, and used this calculus to model objects and concepts found in object-oriented programming languages such as classes, inheritance, and mixins (chapter 8) as well as compositional abstractions (chapter 9). In this chapter, we conclude Part III with a summary of the main observations and lessons learned.

Composition calculus. In order to overcome the problems of existing systems and languages, where the semantics of components and composition is defined in an ad-hoc way, we identified the need for a rigorous semantic foundation (i.e. a *composition calculus*) for specifying applications as compositions of software components. The simplest foundation that seems appropriate for our needs is that of communicating, concurrent agents. The π-calculus, a name-passing calculus in which the topology of communication can evolve dynamically during evaluation [MPW92], addresses shortcomings of other process calculi such as CSP [Hoa85] or CCS [Mil89], and fulfills our basic requirements. In addition, several variants of the basic (monadic) π-calculus can be faithfully encoded in the basic calculus [Mil91, San93], so that it is possible to use the features of richer variants knowing that their meaning can always be understood in terms of the basic π-calculus.

FORM **calculus.** One of the key features of the FORM calculus is the replacement of tuple communication by the communication of extensible record or *forms*. Communicating forms overcomes the restricted extensibility of the tuple-based communication of the polyadic π-calculus [Mil91] as it enables the identification of parameters by *names* rather than positions. This approach makes it much easier to define flexible and extensible, hence reusable abstractions than is possible in the polyadic π-calculus. Since the contents of communications are now independent of positions, agents are more naturally polymorphic as communication forms can be easily extended, and environmental arguments (such as communication policies or default I/O-services) can be passed implicitly.

By introducing the concept of a binary matching operator, which can be used to check for the presence of a binding in a given form, the FORM calculus goes beyond the primitives found in the π-calculus. Label matching in the FORM calculus is similar to name matching in the π-calculus, but in contrast to name matching, two continuation agents (for positive and negative match) can be specified. In contrast to other variants of the π-calculus, where boolean constants have to be encoded as processes, the usage of matching enables the encoding of booleans as values, simplifies the encoding of boolean operations, and allows for a more natural encoding of an "if then else" construct and input-guarded choice.

Although the FORM calculus makes a few fundamental modifications to the π-calculus, it is important to note that it is possible to translate FORM calculus agents to π-calculus processes and back, while preserving behavioural equivalence both ways. This means that any theoretical results that hold in the π-calculus will also hold for the FORM calculus.

Basic composition operations. Our experiments have shown that polymorphic form extension is the basic composition operation for forms. This is of particular interest as forms represent the interfaces (i.e. the set of provided services) of arbitrary components. Polymorphic form extension allows a user to express composition of the services of a given set of components in a flexible, extensible, and robust way.

However, there are situations where polymorphic extension alone is not enough to express the desired behaviour. These situations occur when more than one component offers a service under the same name or if a client should not obtain access to all available services of a component. In these situations, a combination of polymorphic extension and restriction can be used. In particular, the usage of polymorphic form restriction increases the possibilities to control service composition (i.e. extending the default overriding rules of polymorphic extension), without losing the extensibility properties of polymorphic form extension.

Glue abstractions. In situations where compositional mismatches due to incompatible interfaces have to be resolved, glue abstractions can substantially benefit from the keyword-based argument passing of forms. Whereas glue abstractions based on positional parameters hard-wire the corresponding adaptation protocol and, therefore, are less open for extension and adaptation, approaches based on keyword-based parameters are more generic since they make less restrictions on the required interfaces of the components involved. This is of particular interest in the context of adapting and/or extending component interfaces using wrappers. Furthermore, the explicit binding of services to names reduces performance problems of common wrapping techniques as it is possible to avoid a level of indirection and to give direct access to services which are not affected by a wrapper. Finally, wrapper abstractions based on polymorphic form extension help to reduce interface duplication problems of common wrapping techniques discussed in section 5.4.

Object models. The essentials of concurrent objects in the FORM calculus are captured by the basic object model of Pierce and Turner: encapsulation, identity, persistence, instantiation, and synchronization. Extending this model with further abstractions does not change the primary structure: an object is still viewed as an agent containing a set of local agents and channels representing methods and instance variables, respectively, whereas the interface of an object is a form containing bindings for the channels of all exported features. As a result, we obtain an imperative object model with an early binding of self, data encapsulation, and low-level synchronization.

Representing classes as first-class entities (i.e. *class metaobjects*) allows us to extend the basic object model with features such as class variables and methods, inheritance, reusable synchronization policies, and different method dispatch strategies in a natural way. This is achieved by introducing intermediate objects (i.e. objects with an unbound self-reference) specifying the behaviour of new instances, generator agents defining compositions of intermediate objects, and wrappers, which apply a fixed-point operator over composed intermediate objects. Different method dispatch strategies are obtained by applying fixed-point operators at different stages of intermediate object composition.

The concepts of generators, wrappers, and composition of intermediate objects cannot only be used to define classes and class abstractions, but also to model mixins, mixin application, and mixin composition. In contrast to object models found in many object-oriented programming languages, the meta-level abstractions of the FORM calculus-based object model make a clearer separation between functional elements (i.e. methods) and their compositions (i.e. inheritance), which, by plugging together appropriate meta-level functionality, allows us to define multiple object models supporting different kinds of inheritance and method dispatch strategies. It is important to note that keyword-based parameters, polymorphic form extension and restriction are the key concepts to achieve the resulting flexibility and extensibility.

The correct behaviour of the generators defined in the meta-level abstractions for classes and mixins heavily depend on *closures*: the body of a generator method explicitly refers to the parameters passed to the meta-level abstraction used to create the corresponding metaobject.

Composition language. The simple unifying model of agents and forms provided by the FORM calculus enables us to understand how different component models and composition mechanisms can interact, and it provides a tool for experimenting with glue and coordination abstractions that bridge compositional mismatches. Our experiments have shown that a process calculus like the FORM calculus is well-suited for modelling various kinds of components and compositional abstractions. The results of these experiments have influenced PICCOLA(F), an experimental programming language currently under development. The main idea behind the development of PICCOLA(F) is to define all language features by transformation to a core language that implements the FORM calculus, which allows us to understand the properties of higher-level abstractions in terms of the primitives of the underlying calculus.

Part IV

Conclusions

Chapter 11

Conclusions and future work

This chapter brings this thesis to a close. We summarize the main contributions of this work and conclude with a discussion on future directions in the development of a composition calculus and a general-purpose composition language.

11.1 Conclusions

In this thesis, we have illustrated that

> Making a clear separation between computational elements and their relationships enhances the flexibility, maintainability, and robustness of software systems. This concept can be most naturally expressed in terms of a formal foundation that includes asymmetric record concatenation and restriction.

We have surveyed some of the problems with object-oriented technology – as it is used today – and argued that the flexibility and adaptability needed for applications to cope with changing requirements can be substantially enhanced if we do not only think in terms of *components*, but also in terms of *architectures*, *scripts*, and *glue*. In particular, we claim that open systems development is best supported by consciously applying the paradigm

Applications = Components + Scripts.

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. Components are not used in isolation, but according to a software architecture which determines the interfaces that components may have, and the rules governing their composition.

Scripts encapsulate how the components are composed and help to make a clear separation between computational elements and their relationships. By collecting this information in a single location (rather than distributing it amongst the components), we make the architecture of an application easier to understand and maintain. By focusing on building applications as compositions of components, we encourage the use of simple, plug-compatible interfaces. Finally, by emphasizing composition (rather than inheritance), we encourage the reuse of component frameworks and the corresponding architectural styles.

The approach of using components and scripts for open systems development is probably best underlined by the following quote:

> What I think is quite important, but often underrated, is the dichotomy that scripting forces on application design. It encourages the development of reusable components (i.e. "bricks") in system programming languages and the assembly of these components with scripts (i.e. "mortar"). *Brent Welch*

Although software architectures and coordination languages also focus on separating concerns, the separation of computational elements and their relationships can probably be seen best when we consider scripting languages. Whereas conventional programming languages are perfectly suitable for *implementing* software components, scripting languages are designed for *configuring and connecting* components. Furthermore, scripting languages are *extensible* as new abstractions can be added to the language, encouraging the integration of legacy code into frameworks and applications. Scripting languages typically support a single, specific architectural style of composing components, and they are tailored for a specific application domain.

We have summarized the features and properties of several scripting languages and identified two essential concepts for scripting: *encapsulation and wiring* and a *foreign code concept*. These two features can be deduced from the main purpose of scripting (i.e. connecting components) and are found in any scripting language. Furthermore, we have analyzed the essence of "eval" features, a popular mechanism of several scripting languages for executing dynamically created code, and came to the conclusion that any eval-like feature should be considered as a *string-based interface* to a reusable and adaptable *interpreter component*.

We have presented the FORM calculus, an offspring of the asynchronous π-calculus, as a formal foundation for a composition language. The definition of the FORM calculus is motivated by a set of requirements illustrated in section 2.5. In particular, a formal foundation helps to specify different object and component models, to integrate different compositional abstractions (e.g. synchronization, incremental derivation), and to explore notions of contracts and type compatibility for concurrent systems.

The FORM calculus replaces the tuple communication of the π-calculus by the communication of forms, a special notion of extensible records. This approach overcomes the problem of position-dependent arguments, since the contents of communications are now independent of positions and, therefore, makes it easier to define flexible, extensible, and robust abstractions. Furthermore, the FORM calculus integrates polymorphic form extension and restriction as basic composition mechanisms on forms. Various experiments have shown that a combination of these two mechanisms, together with keyword-based parameters and a binary matching operator, provides a powerful mechanism for software composition, since it allows a user to express the composition of the services of a given set of components in a more flexible way. To our knowledge, the FORM calculus is the first process calculus which integrates forms, polymorphic form extension and restriction as well as a binary matching operator as primitives.

The development of concurrent object-based programming languages has suffered from the lack of any generally accepted formal foundation for defining their semantics, although several formal models have been proposed (refer to [Men94] for a summary). Most of these models define objects and object-oriented abstractions as primitives, but they either hard-wire the underlying inheritance model [Red88], integrate concepts in a non-orthogonal way [FKF98], or do not incorporate important features found in object-oriented programming languages (e.g. they lack inheritance [ACV96]).

The FORM calculus, on the other hand, does not integrate objects and object-oriented abstractions as primitives, but provides the required mechanisms to define various object (and component) models. In fact, the FORM calculus can be viewed as a kind of workbench for evaluating and implementing object models, which facilitates the mediation between different object and component models. One of the main insights in using the FORM calculus for modelling objects is the fact that it allows for a compositional view of features found in object-oriented programming languages and, therefore, makes a stronger separation between functional elements (i.e. methods) and their composition. Hence, inheritance is not a fundamental composition mechanism, but is simply an application of object composition.

Summarizing the main observations of Part III, we can say that the FORM calculus facilitates the specification of both concurrent objects and compositional abstractions, and clarifies various concepts which are typically merged (or confused) in existing programming languages. Although our experiments are neither exhaustive nor canonical, we think that the results represent to a certain degree the essence of component-based software development.

11.2 Future work

In this section, we suggest some possibilities for future developments in the area of a composition language in general and a composition calculus in particular.

Composition calculus. The FORM calculus integrates both the notions of agents and forms as primitives. However, the similarity of these two concepts suggests an opportunity for unifying the two concepts. Is it possible to simplify the calculus by viewing an agent as an expression that evaluates to a form? Furthermore, as we have shown in section 7.3.3, the FORM calculus allows us to encode boolean constants as values. However, we could have used a process-based encoding of booleans instead, similar to the one presented for the π-calculus in section 7.1. These observations somehow indicate that there are redundant mechanisms in the calculus, although the primitives of the FORM calculus are all orthogonal (i.e. none of them can be expressed as a combination of other primitives). Therefore, the question arises whether there is something more fundamental behind the notions of agents and forms, which has the same expressive power, flexibility, and extensibility as the FORM calculus.

As we have outlined in section 8.7, there are situations where the expressive power of the FORM calculus is not enough, and we illustrated compositional abstractions which require information about all labels of a given form. Therefore, another question arises whether the FORM calculus should be extended in a way that *labels are first-class values*. First-class labels would make polymorphic extension obsolete as it could be replaced by an abstraction (based on binding extension and full label introspection) with the same behaviour. This, however, suggests that a calculus with labels as first-class values could be a first step into the direction of a unifying concept for agents and forms.

In [PS95], Parrow and Sangiorgi present complete axiomatisations of equivalences for several variants of the π-calculus. They claim that these axiomatisations are applicable for other calculi where value-passing is a basic construct and where logical tests can be made on identity of the values. Therefore, it should be possible to adapt the presented axiomatisations to the FORM calculus as well. This would help us to clarify the completeness and correctness of the labelled transition system as well as the bisimulation relation presented in sections 7.3.7 and 7.3.8, respectively.

Type system. The fundamental purpose of a type system is to prevent the occurrence of *run-time* errors when executing a program, as types impose constraints which help to enforce the correctness of a program [CW85]. This is of particular importance in the context of composition as we would like to detect compositional mismatches based on mismatched types as early as possible. Furthermore, our previous work in modelling objects in the π-calculus has revealed that, besides scoping rules, a type system is an important mechanism for controlling the visibility of features: type restriction can be used to hide features, whereas type extension allows us to add and/or redefine features [LSN96, SL97]. Therefore, it is a natural step to define an appropriate type system for the FORM calculus.

Lumpe has presented a first-order type system for the $\pi\mathcal{L}$-calculus that incorporates asymmetric record concatenation [Lum99]. The presented type system does not incorporate parametric polymorphism, which is supported by the type system of PICT [Tur96]. However, our previous work in modelling objects in PICT has shown that parametric polymorphism is an essential feature for genericity [LSN96, SL97], and several of the compositional abstractions we have illustrated in this work (e.g. the class abstractions presented in section 8.3) cannot be statically type-checked without using parametric polymorphism. Hence, a type system for the FORM calculus as well as a type inference algorithm should integrate parametric polymorphism.

Typing polymorphic form restriction presents similar problems to those encountered in typing polymorphic form extension [Lum99]. Due to the subsumption rule, we must ensure that if $F\backslash X_1$ is well-typed and X_2 is a subtype of X_1, then $F\backslash X_2$ is well-typed and can be used in any context where $F\backslash X_1$ can be used. Hence, an interesting question arises what kind of typing rules we have to use in order to statically type-check polymorphic form restriction and what kind of constraints we have to specify for both arguments.

Another aspect in the context of a composition calculus related to typing is the notion of *component substitutability* [ND95]. Traditional approaches of substitutability (e.g. subtyping, matching) are based on the types of *interfaces*. However, such an approach is generally not applicable when components have their own thread of control and may delay the servicing of requests according to synchronization constraints. In order to overcome the problems of interface-based component substitutability, Nierstrasz has proposed a type framework that characterizes objects and components as regular finite state processes [Nie95], and it would be an interesting research topic to adapt and integrate this approach in a type system for the FORM calculus. The corresponding results would also answer the question how dynamic aspects of components can be expressed in a type system.

The concept of forms defined in the FORM calculus is suitable for specifying the *provided* services of components. However, our modellings do not address the question how *required* services of components can be expressed using forms. Since a type system for a composition language must be able to verify the correct wiring of the provided and required services of all components of an application, it is essential that we have the possibility to specify both the provided and required services of components.

Distribution. In the field of concurrent and distributed systems, various process calculi have recently been proposed that incorporate other aspects of distributed computation, such as communication failure, distributed scopes, and security [CG98, FG96, VC98]. In our work, we have focussed on composed systems within a single administrative domain and do not take into account other distribution aspects. It is obvious, however, that a composition calculus that properly addresses these aspects will play a crucial role for modelling composition of distributed components.

Object models. Although meta-level approaches for modelling objects are usually associated with a *metaobject protocol* (MOP), the meta-level framework presented in section 8.6 only incorporates a basic metaobject protocol. Therefore, two main questions arise: what kind of MOP is needed for a composition language, in particular for mediating between various object models, and what kind of type system is necessary in order to type-check the resulting MOP? The meta-level framework defined in this work goes a long way to support the required flexibility and extensibility, but additional experiments with different kinds of class models are needed in order to end up with a suitable framework that also incorporates aspects such as distribution and persistence. Furthermore, most programming languages that support a run-time MOP are not statically typed, and it would be a challenging work to investigate static type-checking for run-time metaobject protocols, especially in the context of keyword-based parameters as well as asymmetric record concatenation and restriction.

Composition language. Ultimately, we are targeting the development of open, hence distributed systems. The overall goal of our work is the development of a formal model for software composition, integrating a black-box framework for modelling objects and components, and an executable composition language for specifying applications as compositions of software components.

Throughout Part III, we have used PICCOLA(F), an experimental programming language currently under development at our institute, as an executable specification language. One of the reasons of using PICCOLA(F) was to explain our modelling steps at a higher level of abstraction, as using the plain FORM calculus is like programming in a concurrent assembler. Furthermore, we hope that our experiments give us insights what kind of abstractions are needed for software composition and to base the first version of a composition language on the main results of our prototype implementation. The results we described in this work show that a composition language can be directly built on top of a unifying foundation of agents and forms. Before a first version of such a language can be defined, further aspects have to be considered (definition of a higher-level syntax, foreign code concept, distributed and concurrent run-time system to name just a few). However, a detailed discussion of all aspects is beyond the scope of this work.

Methodological issues. In this thesis, we have focussed mainly on technological issues, but there are just as many, and arguable equally important, *methodological issues*: component frameworks focus on software solutions, not problems, so *how can we drive analysis and design* so that we will arrive at the available solutions? Frameworks are notoriously hard to develop, so *how can we iteratively evolve existing object-oriented applications* in order to arrive at a flexible component-based design? Given a problem domain and a body of experience from several applications, how do we re-engineer the software into a component framework? As we develop a component framework, how do we select a suitable architectural style to support black-box composition? Finally, and perhaps most important, software projects are invariably focussed toward the bottom line, so *how can we convince management* to invest in component technology?

Although we do not pretend to have the answers to all the questions raised in this section, we believe that separating applications into components and scripts (i.e. making a clear separation between computational elements and their relationships) is a necessary step towards a methodology for component-based software development.

Part V

Appendices

Appendix A

Source code of UniBE phone book application

In the following, we list the complete source code of the stream framework (section A.1), the string dictionaries used (section A.2) as well as the HTML-page parsing filter `ParseAddr` (section A.3). For the source of the `ubtb` and the top-level function `main`, refer to Figure 6.2.

A.1 Stream framework

```
class Stream:
    '''
    An abstract class supporting streams of items.  next() returns the
    next item, or None, if we are at the end of the stream.
    '''
    def next(self):
        '''Returns next item, or None'''
        raise "Subclass responsibility!"
    def asList(self):
        '''Returns contents of stream as a list.'''
        contents = []
        for item in self:
            contents.append(item)
        return contents

    def __or__(self, target):
        '''Pipe:  compose stream with target function or input stream.'''
        return target(self)
    def __getitem__(self, key):
        '''Allow streams to be used in for loops.'''
        next = self.next()
        if next is None:
            raise IndexError
        else:
            return next
```

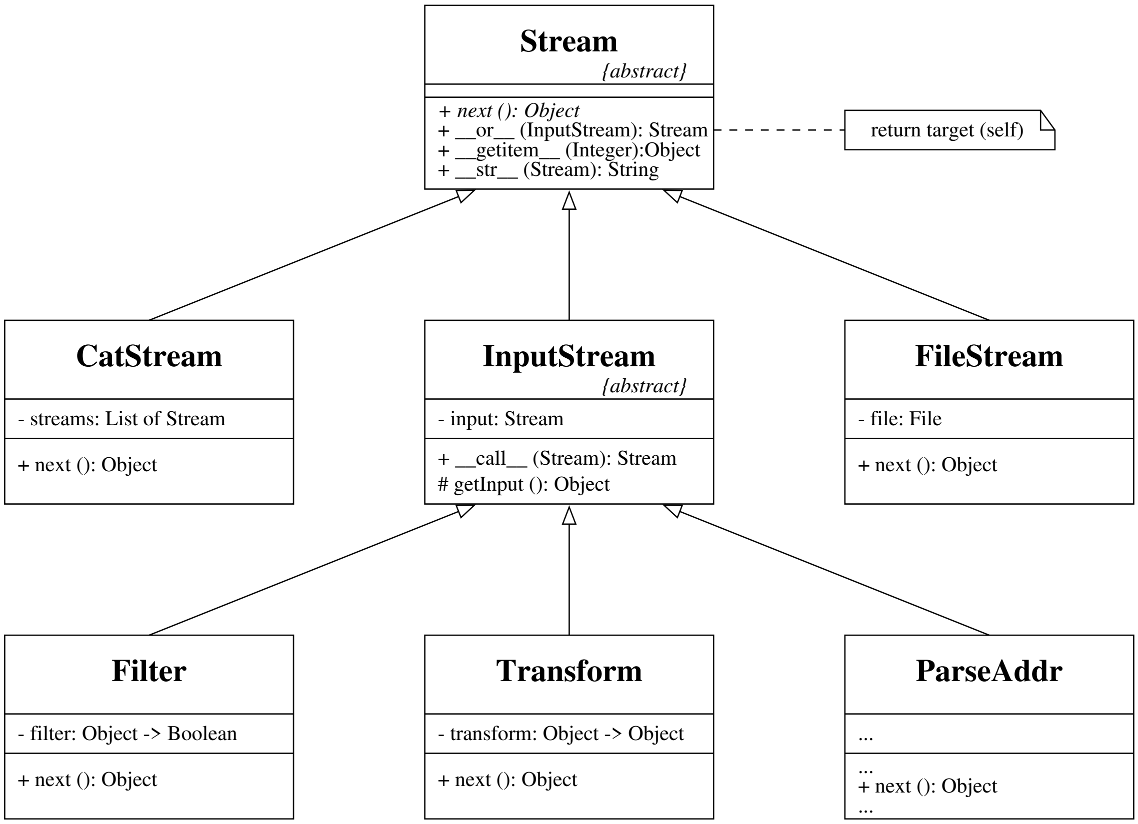


Figure A.1: UML diagram of stream framework.

```
    def __str__(self):
        '''Return a string representation of the stream contents.'''
        contents = []
        for item in self:
            contents.append(item)
        return string.join(contents, '')

    def __add__(self, arg):
        '''Concatenate an argument stream to the end of this one.'''
        return CatStream([self, arg])

# ======================================================================== #

class InputStream(Stream):
    '''
    An abstract class supporting pipes and filters composition of streams.
    A Stream can be piped into an InputStream, causing the stream to be
    bound to input variable of the InputStream.  The next() method of the
    InputStream should normally be defined in terms of getInput().
    '''
```

```
    def __call__(self, input):
        self._input = input
        return CompositeStream(input, self)
    def getInput(self):
        return self._input.next()

# ======================================================================= #

class CompositeStream:
    '''
    Helper class for composing input streams with a possibly still unbound
    input:  '__call__' is forwarded to the lefthand stream, all other
    methods to the righthand stream.
    NOTE: not a subclass of 'InputStream', but offers same interface...
    '''
    def __init__(self, left, right):
        self._left = left
        self._right = right

    def __getattr__(self, name):
        if name == "__call__":
            return getattr(self._left, name)
        else:
            return getattr(self._right, name)

# ======================================================================= #

class CatStream(Stream):
    '''
    Concatenate a list of streams.
    '''
    def __init__(self, streams):
        self._streams = streams

    def next(self):
        '''Return the first item available from the first non-empty stream'''
        while 1:
            if len(self._streams) == 0:
                return None
            next = self._streams[0].next()
            if next is None:
                # The first stream is empty, so move to the next one:
                self._streams = self._streams[1:]
            else:
                return next

    def __add__(self, arg):
        return CatStream([self, arg])

# ======================================================================= #

class FileStream(Stream):
```

```
    '''
    Wrap a file as a stream.
    '''
    def __init__(self, file):
        self._file = file
    def next(self):
        item = self._file.readline()
        if len(item) == 0:
            return None
        else:
            return item

# ========================================================================== #

class Filter(InputStream):
    '''
    Filters a stream by some predicate.  A filter must be used in a pipes
    and filters chain.  A filter is first instantiated with
    just the filter function.  The resulting function object is treated
    as the target of a pipe, causing the stream to be bound.
    '''
    def __init__(self, filter):
        self._filter = filter

    def next(self):
        '''return next element in stream that matches filter predicate.'''
        while 1:
            next = self.getInput()
            if next is None:
                return None
            elif self._filter(next):
                return next

# ========================================================================== #

class Transform(InputStream):
    '''
    Transforms a Stream by some function.  As with Filter, must be used in
    a pipes and filters chain by early-binding the transform function
    and late-binding the stream.
    '''
    def __init__(self, transform):
        self._transform = transform

    def next(self):
        '''Return the next element in the stream.'''
        next = self.getInput()
        if next is None:
            return None
        else:
            return self._transform(next)
```

A.2 String dictionaries

```
class FmtDict(UserDict):
    '''
    A wrapper around a dictionary that has some smarts about how
    to display its fields.
    '''
    def __init__(self):
        UserDict.__init__(self)

    def display(self, fields=None, padding=20):
        '''Display key value pairs as formatted text'''
        if fields is None:
            fields = self.fields # must be defined by subclasses
        lines = []
        for key in fields:
            if self.has_key(key):
                lines.append(("%%-%ds%%s"%padding) %(key+':', self[key]))
        return string.join(lines, '\n') + '\n'

    def select(self, fields):
        '''Return the list of values associated with a list of keys.'''
        return map(lambda k, d=self:  d.get(k,''), fields)

    def delText(self, fields=None):
        '''Display values as a single line of a delimited text table.'''
        if fields is None:
            fields = self.fields
        return joinFields(self.select(fields))

# ======================================================================== #

def joinFields(fields):
    '''Join a list of fields into tab-separated values.'''
    return '"%s"' % string.join(fields, '"\t"')

# ======================================================================== #

class Address(FmtDict):
    '''A FmtDict that knows which address fields are of interest.'''
    fields = [ 'Title', 'Name', 'Institute', 'Address',
            'Phone', 'Alternate phone', 'Email', 'Personal URL' ]
```

A.3 HTML parsing filter

```
class ParseAddr(InputStream):
    '''
    Parse input lines and emit Address objects.
    '''
    def __init__(self):
        # Pre-compile a couple of regular expressions ...
```

```
        import re
        # Recognize(key, value) pairs:
        self._getFields = re.compile(r'^<strong>([^<]*)</strong>\s*(.*)$')
        # Get rid of the HTML anchors surrounding text:
        self._stripAnchor = re.compile(r'^<a href=[^>]+>([^<]+)</a>$')
        self._prevKey = None

    def next(self):
        addr = Address() # Create an empty Address object
        self.startAddr() # Find the start of the address
        while 1:
            line = self.getInput()
            if line is None or line[:13] == " <p><hr>":
                # We reached the end of this address
                if addr:
                    return addr
                else:
                    return None
                (key, val) = self.gf(line)
            if key:
                if addr.has_key(key):
                    # A continuation line, so extend last value
                    addr[key] = "%s, %s" %(addr[key], val)
                else:
                    addr[key] = val

    def startAddr(self):
        '''Queue to the start of an Address to parse(or eof).'''
        while 1:
            line = self.getInput()
            if line is None or line[:10] == "</pre><h2>":
                return

    def gf(self, line):
        '''Get(key,value) fields, if present in line.'''
        mo = self._getFields.match(line)
        if mo:
                (key,val) = mo.group(1,2)
            val = self.sa(val)
            if key == "":
                key = self._prevKey
            self._prevKey = key
            return(key, val)
        else:
            self._prevKey = None
            return(None, None)

    def sa(self, val):
        '''Strip off the HTML anchor from a string value.'''
        return self._stripAnchor.sub(r'\1', val)
```

Appendix B

Congruence of weak bisimulation

In this appendix, we present the proofs concerning $\overset{\mathcal{F}}{\approx}$ being a congruence relation, which have been omitted in section 7.3.8.

Lemma B.1 *For every agent A and $\sigma = \{F/X\}$ with $\mathrm{fv}(A) = \emptyset$, it holds that*

$$A\sigma = A.$$

PROOF: By induction on the structure of A. $\square$

Proposition B.1 *Let A be a closed agent (i.e. $\mathrm{fv}(A) = \emptyset$) and μ be an action. Then $A \overset{\mu}{\Longrightarrow} A'$ implies $\mathrm{fv}(A') = \emptyset$ (i.e. a closed agent evolves to a closed agent).*

PROOF: We proceed by induction on the structure of A. We consider the most significant case $A = a(X).A_1$ (the other cases are similar).

Then we have $A \overset{a(\langle\widetilde{l\!=\!b}\rangle)}{\Longrightarrow} A_1\{\langle\widetilde{l\!=\!b}\rangle/X\}$. Furthermore, $\mathrm{fv}(A) = \emptyset$ implies that the set of free variables $\mathrm{fv}(A_1)$ can be at least a singleton (i.e. $\mathrm{fv}(A_1) = \{X\}$), since the communication removes the binder for X. By definition $\mathcal{V}(\langle\widetilde{l\!=\!b}\rangle) = \emptyset$. Therefore, $A_1\{\langle\widetilde{l\!=\!b}\rangle/X\}$ does not add any free form variables, which implies that $\mathrm{fv}(A_1\{\langle\widetilde{l\!=\!b}\rangle/X\}) = \emptyset$ as required. $\square$

Proposition B.2 $\overset{\mathcal{F}}{\approx}$ *is an equivalence relation.*

PROOF: Symmetry is by definition, while reflexivity is immediate. The only nontrivial property to show is transitivity. We show that the relation ($\overset{\mathcal{F}}{\approx} \circ \overset{\mathcal{F}}{\approx}$) is a weak $\mathcal{F}$-bisimulation. Suppose that

$$A \overset{\mathcal{F}}{\approx} \circ \overset{\mathcal{F}}{\approx} C.$$

Then for some agent B we have

$$A \overset{\mathcal{F}}{\approx} B \text{ and } B \overset{\mathcal{F}}{\approx} C.$$

Consider the case of τ or *output actions* with $\mathrm{bn}(\mu) \cap \mathrm{fn}(A|B|C) = \emptyset$:

- Let $A \overset{\mu}{\Longrightarrow} A'$ and μ either τ or output. Since $A \overset{\mathcal{F}}{\approx} B$, for some agent B' it follows that $B \overset{\mu}{\Longrightarrow} B'$ and $A' \overset{\mathcal{F}}{\approx} B'$.
- Since $B \overset{\mathcal{F}}{\approx} C$, we have for some agent C', $C \overset{\mu}{\Longrightarrow} C'$ and $B' \overset{\mathcal{F}}{\approx} C'$.

Hence, $A' \overset{\mathcal{F}}{\approx} \circ \overset{\mathcal{F}}{\approx} C'$. Similarly, if $C \overset{\mu}{\Longrightarrow} C'$ we can find an agent A' such that $A \overset{\mu}{\Longrightarrow} A'$ and $A' \overset{\mathcal{F}}{\approx} \circ \overset{\mathcal{F}}{\approx} C'$.
Consider the case of composition with output:

- If $A \overset{\mathcal{F}}{\approx} B$, then for all messages $\overline{a}(\langle \widetilde{l = b} \rangle)$ we have by definition $A \mid \overline{a}(\langle \widetilde{l = b} \rangle) \overset{\mathcal{F}}{\approx} B \mid \overline{a}(\langle \widetilde{l = b} \rangle)$.
- Since $B \overset{\mathcal{F}}{\approx} C$, for all messages $\overline{a}(\langle \widetilde{l = b} \rangle)$ we have by definition $B \mid \overline{a}(\langle \widetilde{l = b} \rangle) \overset{\mathcal{F}}{\approx} C \mid \overline{a}(\langle \widetilde{l = b} \rangle)$.

Hence, $A \mid \overline{a}(\langle \widetilde{l = b} \rangle) \overset{\mathcal{F}}{\approx} \circ \overset{\mathcal{F}}{\approx} C \mid \overline{a}(\langle \widetilde{l = b} \rangle)$. Similarly if we start with C. $\square$

Proposition B.3 *For any agents A and B and name c:*

$$A \overset{\mathcal{F}}{\approx} B \;\Rightarrow\; (\nu\ c)A \;\overset{\mathcal{F}}{\approx}\; (\nu\ c)B.$$

PROOF: We show that the relation

$$\mathcal{R} = \{\ ((\nu\ c)A, (\nu\ c)B) \ \mid\ A \overset{\mathcal{F}}{\approx} B\ \} \cup \overset{\mathcal{F}}{\approx}$$

is a weak $\mathcal{F}$-bisimulation.
Consider τ or *output actions* with $\mathrm{bn}(\mu) \cap \mathrm{fn}(A|B) = \emptyset$:

> $(\nu\ c)A \overset{\mu}{\Longrightarrow} (\nu\ c)A'$ is inferred from $A \overset{\mu}{\Longrightarrow} A'$. Since $A \overset{\mathcal{F}}{\approx} B$, this implies that $B \overset{\mu}{\Longrightarrow} B'$ with $A' \overset{\mathcal{F}}{\approx} B'$. Then $(\nu\ c)B \overset{\mu}{\Longrightarrow} (\nu\ c)B'$ is the required move, since $((\nu\ c)A', (\nu\ c)B') \in \mathcal{R}$.

Consider *input actions*:

> $(\nu\ c)A \overset{a(\langle \widetilde{l = b} \rangle)}{\Longrightarrow} (\nu\ c)A'$ is inferred from $A \overset{a(\langle \widetilde{l = b} \rangle)}{\Longrightarrow} A'$. Since $A \overset{\mathcal{F}}{\approx} B$ and, by definition, $(A|\overline{a}(\langle \widetilde{l = b} \rangle), B|\overline{a}(\langle \widetilde{l = b} \rangle)) \in \mathcal{R}$ for all messages $\overline{a}(\langle \widetilde{l = b} \rangle)$, this implies that $B \overset{a(\langle \widetilde{l = b} \rangle)}{\Longrightarrow} B'$ with $A' \overset{\mathcal{F}}{\approx} B'$. Then $(\nu\ c)B \overset{a(\langle \widetilde{l = b} \rangle)}{\Longrightarrow} (\nu\ c)B'$ is the required move, since $((\nu\ c)A', (\nu\ c)B') \in \mathcal{R}$. $\square$

Proposition B.4 *For any agents A, B, and C it holds that*

$$A \overset{\mathcal{F}}{\approx} B \;\Rightarrow\; A|C \;\overset{\mathcal{F}}{\approx}\; B|C.$$

PROOF: We show that the relation

$$\mathcal{R} = \{\ (A|C, B|C)\ \mid\ A \overset{\mathcal{F}}{\approx} B\ \} \cup \overset{\mathcal{F}}{\approx}$$

is a weak $\mathcal{F}$-bisimulation. Input and output are straightforward. We only show the case for τ transitions. There are two possibilities:

- COM: $A|C \overset{\tau}{\Longrightarrow} A'|C'$ is inferred from $A \overset{\overline{a}(\widetilde{\langle l \mapsto b \rangle})}{\Longrightarrow} A'$ and $C \overset{a(\widetilde{\langle l \mapsto b \rangle})}{\Longrightarrow} C'$. Since $A \overset{\mathcal{F}}{\approx} B$, this implies $B \overset{\overline{a}(\widetilde{\langle l \mapsto b \rangle})}{\Longrightarrow} B'$ with $A' \overset{\mathcal{F}}{\approx} B'$. Then $B|C \overset{\tau}{\Longrightarrow} B'|C'$ with $C \overset{a(\widetilde{\langle l \mapsto b \rangle})}{\Longrightarrow} C'$ is the required move, since $(A'|C', B'|C') \in \mathcal{R}$. We have a similar reasoning if $A \overset{a(\widetilde{\langle l \mapsto b \rangle})}{\Longrightarrow} A'$ and $C \overset{\overline{a}(\widetilde{\langle l \mapsto b \rangle})}{\Longrightarrow} C'$.

- CLOSE: $A|C \overset{\tau}{\Longrightarrow} (\nu\ \tilde{c})(A'|C')$ is inferred from $A \overset{(\nu\ \tilde{c})\overline{a}(\widetilde{\langle l \mapsto b \rangle})}{\Longrightarrow} A'$ and $C \overset{a(\widetilde{\langle l \mapsto b \rangle})}{\Longrightarrow} C'$. Since $A \overset{\mathcal{F}}{\approx} B$, this implies $B \overset{(\nu\ \tilde{c})\overline{a}(\widetilde{\langle l \mapsto b \rangle})}{\Longrightarrow} B'$ with $A' \overset{\mathcal{F}}{\approx} B'$. Then $B|C \overset{\tau}{\Longrightarrow} (\nu\ \tilde{c})(B'|C')$ with $C \overset{a(\widetilde{\langle l \mapsto b \rangle})}{\Longrightarrow} C'$ is the required move, since $(A'|C', B'|C') \in \mathcal{R}$. We have a similar reasoning if $A \overset{a(\widetilde{\langle l \mapsto b \rangle})}{\Longrightarrow} A'$ and $C \overset{(\nu\ \tilde{c})\overline{a}(\widetilde{\langle l \mapsto b \rangle})}{\Longrightarrow} C'$. □

Proposition B.5 *For any agents A and B, name a, and form variable X:*

$$A \overset{\mathcal{F}}{\approx} B \;\Rightarrow\; a(X).A \;\overset{\mathcal{F}}{\approx}\; a(X).B.$$

PROOF: $a(X).A$ can only move on input. Therefore, in the case of an input action, we have $a(X).A \overset{a(\widetilde{\langle l \mapsto b \rangle})}{\Longrightarrow} A\{\widetilde{\langle l\!=\!b \rangle}/X\}$. Since $A \overset{\mathcal{F}}{\approx} B$, this implies that $a(X).B$ has the same move such that $a(X).B \overset{a(\widetilde{\langle l \mapsto b \rangle})}{\Longrightarrow} B\{\widetilde{\langle l\!=\!b \rangle}/X\}$ and $(A\{\widetilde{\langle l\!=\!b \rangle}/X\}, B\{\widetilde{\langle l\!=\!b \rangle}/X\}) \in \mathcal{R}$. Since A and B are closed, i.e. $X \notin \mathrm{fv}(A|B)$, by Lemma B.1 we have $A\{\widetilde{\langle l\!=\!b \rangle}/X\} = A$, $B\{\widetilde{\langle l\!=\!b \rangle}/X\} = B$, and it follows $(A, B) \in \mathcal{R}$. □

In fact, an input prefix in the FORM calculus, unlike the π-calculus, is not a binder for names. Therefore, we have $\mathrm{bn}(A) - \mathrm{bn}(a(X).A) = \emptyset$.

Lemma B.2 $!a(X).A \overset{\mathcal{F}}{\approx} !a(X).A|a(X).A$

PROOF: We show that the relation

$$\mathcal{R} = \{\ (!a(X).A|C, !a(X).A|a(X).A|C)\ \mid\ C\ arbitrary\ \} \cup \overset{\mathcal{F}}{\approx}$$

is a weak $\mathcal{F}$-bisimulation. The case for output is obvious. We only show the case for input. Remaining cases are similar. Assume $!a(X).A|C \stackrel{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} !a(X).A|C'$. But if $C \stackrel{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} C'$ the result trivially holds. If not, we should have

$$\text{(REPL)} \quad \frac{a(X).A \stackrel{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} A\{\widetilde{\langle l=b\rangle}/X\}}{!a(X).A \stackrel{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} A\{\widetilde{\langle l=b\rangle}/X\} \mid !a(X).A}$$

and

$$\text{(PAR)} \quad \frac{!a(X).A \stackrel{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} A\{\widetilde{\langle l=b\rangle}/X\} \mid !a(X).A}{!a(X).A \mid C \stackrel{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} (A\{\widetilde{\langle l=b\rangle}/X\} \mid !a(X).A) \mid C},$$

but then

$$\text{(PAR)} \quad \frac{a(X).A \stackrel{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} A\{\widetilde{\langle l=b\rangle}/X\}}{a(X).A \mid !a(X).A \stackrel{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} A\{\widetilde{\langle l=b\rangle}/X\} \mid !a(X).A}$$

and

$$\text{(PAR)} \quad \frac{a(X).A \mid !a(X).A \stackrel{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} A\{\widetilde{\langle l=b\rangle}/X\} \mid !a(X).A}{(a(X).A \mid !a(X).A) \mid C \stackrel{a(\widetilde{\langle l=b\rangle})}{\Longrightarrow} (A\{\widetilde{\langle l=b\rangle}/X\} \mid !a(X).A) \mid C},$$

hence done. □

Using Lemma B.2, a term $!a(X).A$ can be replaced by an arbitrary number of parallel compositions of $a(X).A$.

Proposition B.6 *For any agents A and B:*

$$a(X).A \stackrel{\mathcal{F}}{\approx} a(X).B \Rightarrow !a(X).A \stackrel{\mathcal{F}}{\approx} !a(X).B.$$

PROOF: We show that the relation

$$\mathcal{R} = \{\ (!a(X).A|C, !a(X).B|D) \ \mid\ a(X).A \stackrel{\mathcal{F}}{\approx} a(X).B \textit{ and } C \stackrel{\mathcal{F}}{\approx} D\} \cup \stackrel{\mathcal{F}}{\approx}$$

is a weak $\mathcal{F}$-bisimulation. Output transitions are easy. For τ transitions, only the transition between $!a(X).A$ and C is not immediate. Therefore, we assume that

$$!a(X).A|C \stackrel{\tau}{\Longrightarrow} !a(X).A|A\{\widetilde{\langle l=b\rangle}/X\}|C'.$$

Then

$$a(X).A|C \stackrel{\tau}{\Longrightarrow} A\{\widetilde{\langle l=b\rangle}/X\}|C'$$

and by $a(X).A \stackrel{\mathcal{F}}{\approx} a(X).B$ and Lemma B.2, there is a D' such that

$$a(X).B|D \stackrel{\tau}{\Longrightarrow} B\{\langle\widetilde{l\!=\!b}\rangle/X\}|D'$$

and $C' \stackrel{\mathcal{F}}{\approx} D'$, which implies that

$$!a(X).B|D \stackrel{\tau}{\Longrightarrow} !a(X).B|B\{\langle\widetilde{l\!=\!b}\rangle/X\}|D'$$

as required. For input, suppose $!a(X).A|C \stackrel{a(\langle\widetilde{l=b}\rangle)}{\Longrightarrow} !a(X).A|A\{\langle\widetilde{l\!=\!b}\rangle/X\}|C$, but then

$$a(X).B \stackrel{a(\langle\widetilde{l=b}\rangle)}{\Longrightarrow} B\{\langle\widetilde{l\!=\!b}\rangle/X\}$$

and $A\{\langle\widetilde{l\!=\!b}\rangle/X\} \stackrel{\mathcal{F}}{\approx} B\{\langle\widetilde{l\!=\!b}\rangle/X\}$, hence

$$!a(X).B|D \stackrel{a(\langle\widetilde{l=b}\rangle)}{\Longrightarrow} !a(X).B|B\{\langle\widetilde{l\!=\!b}\rangle/X\}|D,$$

but by Proposition B.4, $(!a(X).A|A\{\langle\widetilde{l\!=\!b}\rangle/X\}|C, !a(X).B|B\{\langle\widetilde{l\!=\!b}\rangle/X\}|D) \in \mathcal{R}$ as desired. $\square$

Appendix C

PICCOLA(F) language definition

Script ::= ' **module**' *Name* [*Imports*] *Declarations* [*Main*]

Imports ::= ' **load**' *NameList* ' ;'

NameList ::= *Name* [' ,' *NameList*]

Declarations ::= *Declaration* [*Declarations*]

Declaration ::= ' **extern**' *String Name*
' **new**' *NameList*
' **run**' *Agent*
' **procedure**' *Name* ' (' [*Name*] ')' ' **do**' *Agent*
' **value**' *Name* ' =' *Form*
' **function**' *Name* ' (' [*Name*] ')' ' =' *Form*

Main ::= ' **main**' *Agent*

Agent ::= *PrimaryAgent* [' |' *Agent*]

PrimaryAgent ::= ' **null**'
Location ' !' *Form*
Location ' ?' ' (' *Name* ')' ' **do**' *Agent*
Location ' ?*' ' (' *Name* ')' ' **do**' *Agent*
' **let**' *Declarations* ' **in**' *Agent* ' **end**'
' **if**' *BoolExpression* ' **then**' *Agent* [' **else**' *Agent*] ' **end**'
Application
' (' *Agent* ')'
PrimaryForm ' ;' [*Agent*]

BoolExpression ::= *Value Built-in-BoolOperator Value*
’ [’ *Form* ’ <–’ *Label* ’]’

Location ::= *Name*
Variable ’ .’ *Label*
PrimaryForm ’ .’ *Label*

Application ::= *Location* ’ (’ [*Form*] ’)’
PrimaryForm ’ .’ *Identifier* ’ (’ [*Form*] ’)’

Form ::= *SeqForm*

SeqForm ::= *ReturnForm* [’ ;’ *SeqForm*]

ReturnForm ::= [’ **return**’] *RestrictedForm*

RestrictedForm ::= *PrimaryForm* [*Restrictions*]

Restrictions ::= ’ \’ *Form* [*Restrictions*]
’ \’ *Label* [*Restrictions*]

PrimaryForm ::= ’ <’ [*FormElementList*] ’ >’
’ **let**’ *Declarations* ’ **in**’ *Form* ’ **end**’
Application
’ **if**’ *BoolExpression* ’ **then**’ *Form* [’ **else**’ *Form*] ’ **end**’
’ (’ *Form* ’)’
Variable

FormElementList ::= *FormElement* [’ ,’ *FormElementList*]

FormElement ::= *Variable* [*Restrictions*]
Application [*Restrictions*]
Label ’ =’ *Value*
’ **procedure**’ *Name* ’ (’ [*Name*] ’)’ ’ **do**’ *Agent*
’ **function**’ *Name* ’ (’ [*Name*] ’)’ ’ =’ *Form*

Value ::= *Location*
Number
String
Form

Appendix D

Filter algebra

Operands = {IOE, IO, IE, OE, I, O, E, Q, S, $\emptyset$}.
Operators = $\{<, |, >, |\&, >\&\}$.

- Operator '$<$':

$$\begin{array}{llllll} \text{IOE} & < & \text{Q} & \rightarrow & \text{OE} & \\ \text{IE} & < & \text{Q} & \rightarrow & \text{E} & \Longrightarrow \ \text{IX} < \text{Q} \rightarrow \text{X} \\ \text{IO} & < & \text{Q} & \rightarrow & \text{O} & \quad (\textit{binding of Q to I}) \\ \text{I} & < & \text{Q} & \rightarrow & \emptyset & \end{array}$$

- Operator '$>$':

$$\begin{array}{llllll} \text{IOE} & > & \text{S} & \rightarrow & \text{IE} & \\ \text{OE} & > & \text{S} & \rightarrow & \text{E} & \Longrightarrow \ \text{OY} > \text{S} \rightarrow \text{Y} \\ \text{IO} & > & \text{S} & \rightarrow & \text{I} & \quad (\textit{binding of O to S}) \\ \text{O} & > & \text{S} & \rightarrow & \emptyset & \end{array}$$

- Operator '$>\&$':

$$\begin{array}{llllll} \text{IOE} & >\& & \text{S} & \rightarrow & \text{I} & \Longrightarrow \ \text{XOE} >\& \text{ S} \rightarrow \text{X} \\ \text{OE} & >\& & \text{S} & \rightarrow & \emptyset & \quad (\textit{merge of O and E, binding to S}) \end{array}$$

$$\begin{array}{llllll} \text{IO} & >\& & \text{S} & \rightarrow & \text{I} & \Longrightarrow \ \text{XO} >\& \text{ S} \rightarrow \text{X} \\ \text{O} & >\& & \text{S} & \rightarrow & \emptyset & \quad (\textit{binding of O to S}) \end{array}$$

$$\begin{array}{llllll} \text{IE} & >\& & \text{S} & \rightarrow & \text{I} & \Longrightarrow \ \text{XE} >\& \text{ S} \rightarrow \text{X} \\ \text{E} & >\& & \text{S} & \rightarrow & \emptyset & \quad (\textit{binding of E to S}) \end{array}$$

- Operator '$|$':

$$\begin{array}{llllll} \text{IOE} & | & \text{IOE} & \rightarrow & \text{IOE} & \\ \text{OE} & | & \text{IOE} & \rightarrow & \text{OE} & \Longrightarrow \ \text{XOE} \mid \text{IYE} \rightarrow \text{XYE} \\ \text{IOE} & | & \text{IE} & \rightarrow & \text{IE} & \quad (\textit{binding of O and I, merge of error}) \end{array}$$

$$\begin{array}{lllll}
\text{OE} & | & \text{IE} & \rightarrow & \text{E}
\end{array}$$

$$\begin{array}{lllllll}
\text{IOE} & | & \text{IO} & \rightarrow & \text{IOE} & & \\
\text{OE} & | & \text{IO} & \rightarrow & \text{OE} & & \\
\text{IOE} & | & \text{I} & \rightarrow & \text{IE} & & \\
\text{OE} & | & \text{I} & \rightarrow & \text{E} & & \\
\text{IO} & | & \text{IOE} & \rightarrow & \text{IOE} & & \\
\text{O} & | & \text{IOE} & \rightarrow & \text{OE} & \Longrightarrow & \text{XO} \mid \text{IY} \rightarrow \text{XY} \\
\text{IO} & | & \text{IE} & \rightarrow & \text{IE} & & (\textit{binding of O and I, no error merge}) \\
\text{O} & | & \text{IE} & \rightarrow & \text{E} & & \\
\text{IO} & | & \text{IO} & \rightarrow & \text{IO} & & \\
\text{O} & | & \text{IO} & \rightarrow & \text{O} & & \\
\text{IO} & | & \text{I} & \rightarrow & \text{I} & & \\
\text{O} & | & \text{I} & \rightarrow & \emptyset & &
\end{array}$$

- Operator '|&':

$$\begin{array}{lllllll}
\text{IOE} & |\& & \text{IOE} & \rightarrow & \text{IOE} & & \\
\text{IOE} & |\& & \text{IO} & \rightarrow & \text{IO} & & \\
\text{IOE} & |\& & \text{IE} & \rightarrow & \text{IE} & & \\
\text{IOE} & |\& & \text{I} & \rightarrow & \text{I} & \Longrightarrow & \text{XOE} \mid\& \text{ IY} \rightarrow \text{XY} \\
\text{OE} & |\& & \text{IOE} & \rightarrow & \text{OE} & & (\textit{merge of O and E, binding to I}) \\
\text{OE} & |\& & \text{IO} & \rightarrow & \text{O} & & \\
\text{OE} & |\& & \text{IE} & \rightarrow & \text{E} & & \\
\text{OE} & |\& & \text{I} & \rightarrow & \emptyset & &
\end{array}$$

$$\begin{array}{lllllll}
\text{IO} & |\& & \text{IOE} & \rightarrow & \text{IOE} & & \\
\text{IO} & |\& & \text{IO} & \rightarrow & \text{IO} & & \\
\text{IO} & |\& & \text{IE} & \rightarrow & \text{IE} & & \\
\text{IO} & |\& & \text{I} & \rightarrow & \text{I} & \Longrightarrow & \text{XO} \mid\& \text{ IY} \rightarrow \text{XY} \\
\text{O} & |\& & \text{IOE} & \rightarrow & \text{OE} & & (\textit{binding of O to I}) \\
\text{O} & |\& & \text{IO} & \rightarrow & \text{O} & & \\
\text{O} & |\& & \text{IE} & \rightarrow & \text{E} & & \\
\text{O} & |\& & \text{I} & \rightarrow & \emptyset & &
\end{array}$$

$$\begin{array}{lllllll}
\text{IE} & |\& & \text{IOE} & \rightarrow & \text{IOE} & & \\
\text{IE} & |\& & \text{IO} & \rightarrow & \text{IO} & & \\
\text{IE} & |\& & \text{IE} & \rightarrow & \text{IE} & & \\
\text{IE} & |\& & \text{I} & \rightarrow & \text{I} & \Longrightarrow & \text{XE} \mid\& \text{ IY} \rightarrow \text{XY} \\
\text{E} & |\& & \text{IOE} & \rightarrow & \text{OE} & & (\textit{binding of E to I}) \\
\text{E} & |\& & \text{IO} & \rightarrow & \text{O} & & \\
\text{E} & |\& & \text{IE} & \rightarrow & \text{E} & & \\
\text{E} & |\& & \text{I} & \rightarrow & \emptyset & &
\end{array}$$

Glossary

Architectural description language: an *architectural description language* (ADL) is a notation that allows for a precise description and analysis of the externally visible properties of a software architecture, supporting different architectural styles at different levels of abstraction.

Architectural mismatch: *architectural mismatch* stems from mismatched assumptions a component makes about the structure of the system it is to be part of.

Architectural pattern: an *architectural pattern* describes the solution of a particular recurring design problem that arises in specific design contexts. The solution scheme describes the overall structural organization (components, their responsibilities, and the relationships between them), the constraints of its application, and the associated composition and design rules.

Architectural style: an *architectural* style is an abstraction over a set of related software architectures. It defines a vocabulary of component and connector types and a set of rules how components and connectors can be combined.

Component: a software *component* is a static abstraction with plugs and a composable element of a component framework.

Component framework: a *component framework* is a collection of software components with a software architecture that determines the interfaces that components may have and the rules governing their composition.

Component platform: a *component platform* denotes any soft- or hardware a component is built upon.

Compositional mismatch: a *compositional mismatch* occurs whenever it is impossible to successfully interconnect components of a component framework with the available connectors of the framework.

Coordination: *coordination* is the management of dependencies between concurrent and/or distributed components.

Glue: *glue* is the part of an application which overcomes compositional mismatches.

Interoperability: *interoperability* is the ability of software components to communicate and cooperate with each other.

Scripting: *scripting* is a high-level binding technology for component-based systems.

Scripting language: a *scripting language* is a high-level language used to create, customize, and assemble components into a predefined software architecture.

Software architecture: a *software architecture* describes a software (sub-)system as a configuration of components and connectors. A connector connects required ports of a set of components to provided ports of other components. A configuration of components and connectors can be used as a component of another (sub-)system.

Software composition: *software composition* is the process of constructing applications by interconnecting software components through their plugs.

Bibliography

[AAG93] Gregory Abowd, Robert Allen, and David Garlan. Using Style to Understand Descriptions of Software Architecture. In *Proceedings of SIGSOFT '93*, pages 9–20, December 1993.

[AAG95] Gregory Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. Technical Report CMU-CS-95-1111, School of Computer Science, Carnegie Mellon University, Pittsburgh, January 1995.

[ABB$^+$89] Giuseppe Attardi, Cinzia Bonini, Maria Rosaria Boscotrecase, Tito Flagella, and Mauro Gaspari. Metalevel Programming in CLOS. In Stephen Cook, editor, *Proceedings ECOOP '89*, pages 243–256. Cambridge University Press, July 1989.

[ABW98] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Pattern Smalltalk Companion*. Addison-Wesley, 1998.

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[ACS96] Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. On Bisimulations for the Asynchronous π-calculus. Technical Report RR-2913, INRIA Sophia-Antipolis, June 1996.

[ACV96] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An Interpretation of Objects and Object Types. In *Proceedings POPL '96*, pages 396–409. ACM Press, January 1996.

[AEW96] Philipp Ackermann, Dominik Eichelberg, and Bernhard Wagner. Visual Programming in an Object-Oriented Framework. In *Proceedings of Swiss Computer Science Conference*, 1996.

[AGI97] Robert J. Allen, David Garlan, and James Ivers. Formal Modeling and Analysis of Architectural Standards. Submitted for Publication, 1997.

[Ahg86] Gul Ahga. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language*. Oxford University Press, 1977.

[Aks89] Mehmet Aksit. *On the Design of the Object-Oriented Language Sina*. PhD thesis, University of Twente, NL, 1989.

[Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[All97] Robert J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997.

[ALSN99] 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: An Object-Oriented Approach*, chapter 18. Cambridge University Press, 1999. to appear.

[Ame87] Pierre America. POOL-T: A Parallel Object-Oriented Language. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

[AMST93] Gul Agha, Ian Mason, Scott Smith, and Carolyn Talcott. A Foundation for Actor Computation. Technical report, UIUC, 1993.

[Ana89] Paul C. Anagnostopoulos. *Writing Real Programs in DCL*. Digital Press, 1989.

[AP90] Jean-Marc Andreoli and Remo Pareschi. LO and Behold! Concurrent Structured Processes. In *Proceedings OOPSLA/ECOOP '90*, volume 25 of *ACM SIGPLAN Notices*, pages 44–56, October 1990.

[Arb96] Farhad Arbab. The IWIM Model for Coordination of Concurrent Activities. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models*, LNCS 1061, pages 34–56. Springer, April 1996. Proceedings of Coordination '96.

[Arb98] Farhad Arbab. *Manifold Reference Manual*. Department of Software Engineering, CWI, Amsterdam, NL, June 1998.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[AZ96] Davide Ancona and Elena Zucca. An Algebraic Approach to Mixins and Modularity. In Michael Hanus and Mario Rodríguez-Artalejo, editors, *Proceedings of 5th International Conference on Algebraic and Logic Programming (ALP '96)*, LNCS 1139, pages 179–193. Springer, September 1996.

[BC90] Gilad Bracha and William Cook. Mixin-based Inheritance. In Norman Meyrowitz, editor, *Proceedings OOPSLA/ECOOP '90*, volume 25 of *ACM SIGPLAN Notices*, pages 303–311, October 1990.

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[BCP97] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing Object Encodings. In *Proceedings of Theoretical Aspects of Computer Software (TACS '97)*, LNCS 1281, pages 415–438, August 1997. Springer.

[Bea92] Brian W. Beach. Connecting Software Components with Declarative Glue. In *Proceedings of ICSE '92*, pages 120–137, Melbourne, May 1992. ACM Press.

[BI69] Frederick Brooks and Ken Iverson. *Automatic Data Processing*. Wiley, 1969.

[BL92] Gilad Bracha and Gary Lindstrom. Modularity Meets Inheritance. In *Proceedings of International Conference on Computer Languages*, pages 282–290. IEEE Computer Society, April 1992.

[Blo79] Toby Bloom. Evaluating Synchronization Mechanisms. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 24–32, December 1979.

[BMR$^+$96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, May 1996.

[Bor95] Borland International. *Borland Delphi Users Manual*, 1995.

[Bos97] Jan Bosch. Superimposition: A Component Adaptation Technique. Submitted for Publication, 1997.

[Bou78] S.R. Bourne. An Introduction to the UNIX Shell. *Bell System Technical Journal*, 57(6):1971–1990, July 1978.

[Bou92] Gérard Boudol. Asynchony and the π-calculus. Technical Report 1702, INRIA Sophia-Antipolis, May 1992.

[BPS99] Viviana Bono, Amit J. Patel, and Vitaly Shmatikov. A Core Calculus of Classes and Mixins. In Rachid Guerraoui, editor, *Proceedings ECOOP '99*, LNCS 1628, pages 43–66. Springer, June 1999.

[Bri96] Jean-Pierre Briot. An Experiment in Classification and Specialization of Synchronization Schemes. In Kokichi Futatsugi and Satoshi Matsuoka, editors, *Object Technologies for Advanced Software*, LNCS 1049, pages 227–249. Springer, March 1996. Proceedings ISOTAS '96.

[Bru94] Kim B. Bruce. A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics. *Journal of Functional Programming*, 4(2), April 1994.

[BS95] Manuel Barrio Solorzano. *Estudio de Aspectos Dinamicos en Sistemas Orientados al Objecto*. PhD thesis, Universidad de Valladolid, September 1995.

[Car88] Luca Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, 76:138–164, 1988.

[Car95] Luca Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile Ambients. In Maurice Nivat, editor, *Foundations of Software Science and Computational Structures*, LNCS 1378, pages 140–155. Springer, 1998.

[CM94] Luca Cardelli and John C. Mitchell. Operations on Records. In Carl Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994.

[CN96] Paul C. Clements and Linda M. Northrop. Software Architecture: An Executive Overview. Technical Report CMU/SEI-96-TR-003, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, February 1996.

[Com93] Apple Computer. *AppleScript Language Guide*. Apple Technical Library. Addison-Wesley, 1993.

[Coo89] William R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Department of Computer Science, Brown University, Providence, RI, May 1989.

[Cow90] Michail F. Cowlishaw. *The REXX Language: A practical Approach to Programming*. Prentice Hall, 2nd edition, 1990.

[CP94] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, 1994.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Dam94] Laurent Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches*. PhD thesis, Centre Universitaire d'Informatique, University of Geneva, CH, 1994.

[Dam98] Laurent Dami. A Lambda-Calculus for Dynamic Binding. *Theoretical Computer Science*, 192:201–231, February 1998.

[dB72] Nikolas G. de Bruijn. Lambda Calculus Notation with Nameless Dummies. *Indagationes Mathematicae*, 34:381–392, 1972.

[DDN+97] Serge Demeyer, Stéphane Ducasse, Robb Nebbe, Oscar Nierstrasz, and Tamar Richner. Using Restructuring Transformations to Reengineer Object-Oriented Systems. Technical report, University of Bern, Institute of Computer Science and Applied Mathematics, May 1997.

[Dij68] Edsger W. Dijkstra. The structure of the ”THE” multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[DLDR+91] D. Decouchant, P. Le Dot, M. Riveill, C. Roisin, and X. Rousset de Pina. A Synchronization Mechanism for an Object Oriented Distributed System. In *Proceedings of 11th International Conference on Distributed Computing Systems*, pages 152–159. IEEE, May 1991.

[Duc97] Stéphane Ducasse. Réification des schémas de conception: une expérience. In Roland Ducournau and Serge Garlatti, editors, *Proceedings of Langages et Modèles à Objets ’97*, pages 95–110, Roscoff, October 1997. Hermes.

[DW99] Desmond F. D’Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Object Technology Series. Addison-Wesley, 1999.

[FG96] Cédric Fournet and Georges Gonthier. The Reflexive Chemical Abstract Machine and the Join-Calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM, January 1996.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and Mixins. In *Proceedings POPL ’98*, pages 171–183, San Diego, January 1998. ACM Press.

[Fla97] David Flanagan. *JavaScript: The Definitive Guide*. O’Reilly & Associates, 2nd edition, January 1997.

[FM95] Kathleen Fisher and John C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proceedings of FCT ’95*, LNCS 965, pages 42–61. Springer, 1995.

[FM96] Jesse Feiler and Anthony Meadow. *Essential OpenDoc*. Addison-Wesley, 1996.

[Fon98] Jean-Luc Fontaine. Simple Tcl Only Object Oriented Programming. Available at http://www.multimania.com/jfontain/stooop.htm, 1998.

[GAACB95] Cristina Gacek, Ahmed Abd-Allah, Bradford Clark, and Barry Boehm. On the Definition of Software System Architecture. In David Garlan, editor, *Proceedings of ICSE '95 Workshop on Architectures for Software Systems*, pages 85–95, Seattle, April 1995.

[Gam91] Erich Gamma. *Object-Oriented Software Development based on ET++: Design Patterns, Class Library, Tools*. PhD thesis, Institute for Computer Science, University of Zurich, 1991.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, 12(6):17–26, November 1995.

[GG96] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Peer-to-Peer Communications, December 1996.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[GJ98] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts*. Wiley, 1998.

[GR89] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley, September 1989.

[Gri98] Frank Griffel. *Componentware: Konzepte und Techniken eines Softwareparadigmas*. dpunkt, 1998.

[Hel99] Michael Held. Scripting für CORBA. Master's thesis, University of Bern, Institute of Computer Science and Applied Mathematics, April 1999.

[Hew77] Carl Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Artificial Intelligence*, 8(3):323–364, June 1977.

[Hoa85] Charles A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Höl93] Urs Hölzle. Integrating Independently-Developped Components in Object-Oriented Languages. In Oscar Nierstrasz, editor, *Proceedings ECOOP '93*, LNCS 707, pages 36–56. Springer, July 1993.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *Proceedings ECOOP '91*, LNCS 512, pages 133–147. Springer, July 1991.

[IdFCF96] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua – an Extensible Extension Language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[Joh92] Ralph E. Johnson. Documenting Frameworks using Patterns. In *Proceedings OOPSLA '92*, volume 27 of *ACM SIGPLAN Notices*, pages 63–76, October 1992.

[Joh98] Ray Johnson. Tcl and Java Integration. White Paper, Sun Microsystems, January 1998.

[Jon93] Cliff B. Jones. A Pi-Calculus Semantics for an Object-Based Design Notation. In Eike Best, editor, *Proceedings CONCUR '93*, LNCS 715, pages 158–172. Springer, 1993.

[KdRB91] Grégor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kon93] Dimitri Konstantas. *Cell: a Framework for a Strongly Distributed Object Based System*. PhD thesis, Department of Computer Science, University of Geneva, CH, 1993.

[Kon95] Dimitri Konstantas. Interoperation of Object-Oriented Applications. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 69–95. Prentice Hall, 1995.

[Kru95] Philippe B. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, November 1995.

[Lea96] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. The Java Series. Addison-Wesley, October 1996.

[LSN96] Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Using Metaobjects to Model Concurrent Objects with PICT. In *Proceedings of Langages et Modèles à Objets '96*, pages 1–12, Leysin, October 1996.

[LSNA97] Markus Lumpe, Jean-Guy Schneider, Oscar Nierstrasz, and Franz Achermann. Towards a formal composition language. In Gary T. Leavens and Murali Sitaraman, editors, *Proceedings of ESEC '97 Workshop on Foundations of Component-Based Systems*, pages 178–187, Zurich, September 1997.

[Lum99] Markus Lumpe. *A π-Calculus Based Approach to Software Composition*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999.

[Lut96] Mark Lutz. *Programming Python: Object-Oriented Scripting*. O'Reilly & Associates, October 1996.

[LV95] David C. Luckham and James Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, September 1996.

[McA95] Jeff McAffer. Meta-level Programming with CodA. In Walter Olthoff, editor, *Proceedings ECOOP '95*, LNCS 952, pages 190–214. Springer, August 1995.

[McH94] Ciaran McHale. *Synchronization in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance*. PhD thesis, Department of Computer Science, Trinity Collegue, Dublin, Ireland, October 1994.

[McI69] M.D. McIlroy. Mass Produced Software Components. In P. Naur and B. Randell, editors, *Software Engineering*. NATO Science Committee, January 1969.

[McM78] Lee E. McMahon. SED: a Non-interactive Text Editor. *Bell System Technical Journal*, 57(6), August 1978.

[MDK92] Jeff Magee, Naranker Dulay, and Jeff Kramer. Structuring parallel and distributed programs. In *Proceedings of the International Workshop on Configurable Distributed Systems*, March 1992.

[Men94] Tom Mens. A survey on formal models for OO. Technical Report vub-tinf-tr-94-03, Department of Computer Science, Vrije Universiteit Brussel, Belgium, 1994.

[Mes90] José Meseguer. A Logical Theory of Concurrent Objects. In *Proceedings OOPSLA/ECOOP '90*, volume 25 of *ACM SIGPLAN Notices*, pages 101–115, October 1990.

[Mey92] Bertrand Meyer. *Eiffel: the Language*. Prentice Hall, 1992.

[Mey94] Vicki de Mey. *Visual Composition of Software Applications*. PhD thesis, Department of Computer Science, University of Geneva, Switzerland, 1994.

[Mey98] Bertrand Meyer. The Component Combinator for Enterprise Applications. *Journal of Object-Oriented Programming*, 10(8):5–9, January 1998.

[Mic97] Microsoft Corporation. *Visual Basic Programmierhandbuch*, 1997.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil90] Robin Milner. Functions as Processes. In *Proceedings ICALP '90*, LNCS 443, pages 167–180. Springer, July 1990.

[Mil91] Robin Milner. The Polyadic Pi-Calculus: a Tutorial. Technical Report ECS-LFCS-91-180, Computer Science Department, University of Edinburgh, UK, October 1991.

[MM97] Thomas J. Mowbray and Raphael C. Malveau. *CORBA Design Patterns*. Wiley, 1997.

[MMM95] Hafdeh Mili, Fatma Mili, and Ali Mili. Reusing Software: Issues and Research Directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, June 1995.

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[Mog89] Eugenio Moggi. Computational Lambda-Calculus and Monads. In *Proceedings of Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Part I/II. *Information and Computation*, 100:1–77, 1992.

[MS92] Robin Milner and Davide Sangiorgi. Barbed Bisimulation. In Werner Kuich, editor, *Proceedings of ICALP '92*, LNCS 623, pages 685–695. Springer, July 1992.

[MS96] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.

[MT97] Nenad Medvidovic and Richard N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In Mehdi Jazayeri and Helmut Schauer, editors, *Proceedings ESEC '97*, LNCS 1301, pages 60–76, September 1997.

[ND95] Oscar Nierstrasz and Laurent Dami. Component-Oriented Software Technology. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice Hall, 1995.

[Nie92] Oscar Nierstrasz. Towards an Object Calculus. In Mario Tokoro, Oscar Nierstrasz, and Peter Wegner, editors, *Proceedings of ECOOP '91 Workshop on Object-based Concurrent Computing*, LNCS 612, pages 1–20. Springer, 1992.

[Nie95] Oscar Nierstrasz. Regular Types for Active Objects. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995.

[NM95a] Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a Composition Language. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, pages 147–161. Springer, 1995.

[NM95b] Oscar Nierstrasz and Theo Dirk Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2):262–264, June 1995.

[NP96] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. Technical Report 392, Computer Laboratory, University of Cambridge, UK, January 1996.

[NSL96] Oscar Nierstrasz, Jean-Guy Schneider, and Markus Lumpe. Formalizing Composable Software Systems – A Research Agenda. In *Proceedings 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems*, pages 271–282. Chapmann & Hall, 1996.

[NTMS91] Oscar Nierstrasz, Dennis Tsichritzis, Vicki de Mey, and Marc Stadelman. Objects + Scripts = Applications. In *Proceedings Esprit 1991 Conference*, pages 534–552, Dordrecht, NL, 1991. Kluwer Academic Publisher.

[OHE95] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide*. Wiley, September 1995.

[OMG96] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, July 1996.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[Ous98] John K. Ousterhout. Scripting: Higher Level Programming for the 21st Century. *IEEE Computer*, 31(3):23–30, March 1998.

[Pap92] Michael Papathomas. A Unifying Framework for Process Calculus Semantics of Concurrent Object-Oriented Languages. In Mario Tokoro, Oscar Nierstrasz, and Peter Wegner, editors, *Proceedings of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, LNCS 612, pages 53–79. Springer, 1992.

[Par76] David Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[Pre95] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.

[PS95] Joachim Parrow and Davide Sangiorgi. Algebraic Theories for Name-Passing Calculi. *Information and Computation*, 120(2):174–197, 1995.

[PS99] Wolfgang Pree and Hermann Sikora. Java-Architekturkomponenten – Konzepte, Fallstudien & Erfahrungen. In Silvano Maffeis, Fridtjof Toenniessen, and Christian Zeidler, editors, *Erfahrungen mit Java: Projekte aus Industrie und Hochschule*, chapter 8, pages 199–223. dpunkt, 1999.

[PT94] Benjamin C. Pierce and David N. Turner. Simple Type-Theoretic Foundations for Object-Oriented Programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

[PT95] Benjamin C. Pierce and David N. Turner. Concurrent Objects in a Process Calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Theory and Practice of Parallel Programming (TPPP)*, LNCS 907, pages 187–215. Springer, April 1995.

[PT97] Benjamin C. Pierce and David N. Turner. Pict: A Programming Language based on the Pi-Calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, March 1997.

[Pur94] James M. Purtilo. The POLYLITH Software Bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, January 1994.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[Red88] Uday S. Reddy. Objects as closures: Abstract semantics of object oriented languages. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 289–297. ACM, July 1988.

[Ree96] Trygve Reenskaug. *Working with Objects: the OOram Software Engineering Method*. Manning Publications, 1996.

[RFW96] Jonathan G. Rossie, Daniel P. Friedman, and Mitchell Wand. Modeling Subobject-based Inheritance. In Pierre Cointe, editor, *Proceedings ECOOP '96*, LNCS 1098, pages 248–274. Springer, July 1996.

[Riv96] Fred Rivard. Smalltalk: a Reflective Language. In Grégor Kiczales, editor, *Proceedings Reflection '96*, 1996.

[Rog97] Dale Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.

[Sam97] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer, 1997.

[San93] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, Computer Science Department, University of Edinburgh, UK, May 1993.

[San95] Davide Sangiorgi. Lazy functions and mobile processes. Technical Report RR-2515, INRIA Sophia-Antipolis, April 1995.

[San96a] Davide Sangiorgi. A Theory of Bisimulation for the π-calculus. *Acta Informatica*, 33:69–97, 1996. An extract appeared in *Proceedings of CONCUR '93*, LNCS 715, Springer.

[San96b] Davide Sangiorgi. An interpretation of Typed Objects into Typed Pi-calculus. Technical Report RR-3000, INRIA Sophia-Antipolis, September 1996.

[SC97] Mary Shaw and Paul Clements. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In *Proceedings COMPSAC '97*, Washington, August 1997.

[Set89] Ravi Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley, 1989.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996.

[Sha95] Mary Shaw. Architectural Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging. In Mansur Samadzadeh and Mansour Zand, editors, *Proceedings of SIGSOFT Symposium on Software Reusability*, pages 3–6, Seattle, April 1995. ACM Press.

[SL96] Jean-Guy Schneider and Markus Lumpe. Modelling Objects in PICT. Technical Report IAM-96-004, University of Bern, Institute of Computer Science and Applied Mathematics, January 1996.

[SL97] Jean-Guy Schneider and Markus Lumpe. Synchronizing Concurrent Objects in the Pi-Calculus. In Roland Ducournau and Serge Garlatti, editors, *Proceedings of Langages et Modèles à Objets '97*, pages 61–76, Roscoff, October 1997. Hermes.

[SN98] Jean-Guy Schneider and Oscar Nierstrasz. Scripting: Higher-Level Programming for Component-Based Systems. OOPSLA '98 Tutorial Notes, October 1998.

[SN99] Jean-Guy Schneider and Oscar Nierstrasz. Components, Scripts and Glue. In Leonor Barroca, Jon Hall, and Patrick Hall, editors, *Software Architectures – Advances and Applications*, chapter 2, pages 13–25. Springer, 1999.

[SNH95] Dilip Soni, Robert L. Nord, and Christine Hofmeister. Software Architecture in Industrial Applications. In *Proceedings ICSE '95*, pages 196–207, Seattle, April 1995. ACM Press.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991.

[Sun97a] Sun Microsystems. *Java Native Interface Specification*, May 1997.

[Sun97b] Sun Microsystems. *JavaBeans Specification*, July 1997.

[Sun99] Sun Microsystems. *Enterprise JavaBeans Specification*, August 1999.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[Tsi89] Dennis Tsichritzis. Object-Oriented Development for Open Systems. In *Proceedings IFIP '89*, pages 1033–1040. North-Holland, August 1989.

[Tur96] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, Department of Computer Science, University of Edinburgh, UK, 1996.

[TV89] Chris Tomlinson and Singh Vineet. Inheritance and Synchronization with Enabled-Sets. In Norman Meyrowitz, editor, *Proceedings OOPSLA '89*, volume 24 of *ACM SIGPLAN Notices*, pages 103–112, October 1989.

[Ude94] Jon Udell. Componentware. *Byte*, 19(5):46–56, May 1994.

[US87] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In *Proceedings OOPSLA '87*, volume 22 of *ACM SIGPLAN Notices*, pages 227–242, December 1987.

[Var96] Patrick Varone. Implementation of "Generic Synchronization Policies" in PICT. Technical Report IAM-96-005, University of Bern, Institute of Computer Science and Applied Mathematics, April 1996.

[Vas94] Vasco T. Vasconcelos. Typed Concurrent Objects. In Mario Tokoro and Remo Pareschi, editors, *Proceedings ECOOP '94*, LNCS 821, pages 100–117. Springer, July 1994.

[VC98] Jan Vitek and Giuseppe Castagna. Towards a Calculus of Secure Mobile Computations. In Dennis Tsichritzis, editor, *Electronic Commerce Objects*, pages 31–46. Centre Universitaire d'Informatique, University of Geneva, CH, July 1998.

[VdBL89] Jan Van den Bos and Jan Laffra. PROCOL – A Parallel Object Language with Protocols. In Norman Meyrowitz, editor, *Proceedings OOPSLA '89*, volume 24 of *ACM SIGPLAN Notices*, pages 95–102, October 1989.

[VLM96] Marc Van Limberghen and Tom Mens. Encapsulation and Composition as Orthogonal Operators on Mixins: A Solution to Multiple Inheritance Problems. *Object-Oriented Systems*, 3(1):1–30, March 1996.

[vR96] Guido van Rossum. Python Reference Manual. Technical report, Corporation for National Research Initiatives (CNRI), October 1996.

[Wad92] Philip Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2:461–492, 1992.

[Wal95] David J. Walker. Objects in the Pi-Calculus. *Information and Computation*, 116(2):253–271, 1995.

[WCS96] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, 2nd edition, September 1996.

[WvRA96] Aaron Watters, Guido van Rossum, and James Ahlstrom. *Internet Programming with Python*. MIS Press, October 1996.

[WWRT91] Jack C. Wileden, Alexander L. Wolf, William R. Rosenblatt, and Peri L. Tarr. Specification-level Interoperability. *Communications of the ACM*, 34(5):72–87, May 1991.

[YS97] Daniel M. Yellin and Robert E. Strom. Protocol Specifications and Component Adapters. *ACM Transactions on Programming Languages*, 19(2):292–333, March 1997.